Personal Computer PCjr
Hardware Reference
Library

BASIC

By Microsoft Corp.

(**First edition - June 1983)**

This product could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication.

The following paragraph applies only to the United States and Puerto Rico: International Business Machines Corporation provides this manual "as is," without warranty of any kind, either expressed or implied, including, but not limited to, the particular purpose. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this manual at any time.

Products are not stocked at the address below. Requests for copies of this product and for technical information about the system should be made to your authorized IBM Personal Computer dealer.

A Reader's Comment Form is provided at the back of this publication. If the form has been removed, address comments to: IBM Corp., Personal Computer, P.O. Box 1328-C, Boca Raton, Florida 33432. IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligations whatever.

Preface

This book is a reference for both Cartridge and Cassette versions of BASIC for the PCjr.

Throughout this book, the term *BASIC* refers to both versions of Cassette BASIC and Cartridge BASIC.

In order to use this manual, you should have some knowledge of general programming concepts; we are not trying to teach you how to program in this book. The BASIC Tutorial book that was shipped with your system is designed to give you this general BASIC programming knowledge.

How to Use This Book

The book is divided into four chapters plus ten (10) appendixes.

Chapter 1 is a brief overview of the two versions of PCjr BASIC interpreter.

Chapter 2 tells you what you need to know to start using BASIC on your PCjr. It tells you how to operate your computer using BASIC.

Chapter 3 covers a variety of topics which you need to know before you actually start programming. Much of the information pertains to data representation when using BASIC. We discuss filenames here, along with many of the special input and output features available in IBM PCjr BASIC.

Chapter 4 is the reference section. It contains, in alphabetical order, the syntax and meanings of every command, statement, and function in BASIC.

The appendixes contain other useful information, such as lists of error messages, ASCII codes, and math functions; plus helpful information on machine language subroutines, diskette input and output, and communications. You may find "Appendix D. Converting Programs to IBM PCjr BASIC" especially helpful, because it discusses the differences between IBM PCjr BASIC and other BASICs. You will also find detailed information on more advanced subjects for the more experienced programmer.

We suggest you read thoroughly Chapters 2 and 3 to become familiar with BASIC. Then, while you are actually programming, you can refer to Chapter 4 for the information you need about the commands or statements you are using. **Syntax Diagrams**

Each of the commands, statements, and functions described in this book has its syntax described according to the following conventions:

- Words in capital letters are keywords and must be entered as shown. They may be entered in any combination of uppercase and lowercase. BASIC always converts words to uppercase (unless they are part of a quoted string, remark, or DATA statement).
- You must supply any items in lowercase italic letters.
- Items in square brackets ([ ]) are optional.
- An ellipsis (...) indicates an item may be repeated as many times as you wish.
- All punctuation except square brackets (such as commas, parentheses, semicolons, hyphens, or equal signs) must be included where shown.

Let's look at an example:

INPUT[;]["*prompt*";] *variable*[,*variable*]...

This says that for an INPUT statement to be valid, you must first have the keyword INPUT, followed optionally by a semicolon. Then, if you wish, you may include a *prompt* within quotation marks. The *prompt*, must be followed by a semicolon. An INPUT statement must have at least one *variable* You may have more than one *variable* if you separate them with commas.

More detailed information on each of the parameters is included with the text accompanying the diagram. The information for this example is in Chapter 4, under "INPUT Statement."

Related Publications

The following manuals contain related information that you may find useful:

- The IBM PCjr *Guide to Operations* manual
- The IBM Personal Computer *Disk Operating System* manual
- The IBM PCjr *Technical Reference* manual
- The PCjr BASIC Tutorial
- The IBM Personal Computer *BASIC Compiler* manual

Contents

Notes

Chapter 1. The Versions of BASIC

Contents

Notes

The Versions of BASIC

PCjr offers two different versions of the BASIC interpreter:

- Cassette
- Cartridge

The two versions of BASIC are upward compatible; that is, everything that Cassette BASIC does, Cartridge BASIC does plus a little more. The differences between the versions are discussed in more detail below.

The BASIC commands, statements, and functions for both versions of the BASIC interpreter are described in detail in Chapter 4, "BASIC Commands, Statements, Functions, and Variables." Included in each description is a section called **Versions:**, where we tell you which versions of BASIC support the command, statement, or function.

For example, if you look under "CHAIN Statement" in Chapter 4, you will note that it says:

Versions:	Cassette	Cartridge	Compiler
		***	(**)

The asterisks show which versions of BASIC support the statement. This example shows that you can use the CHAIN statement for programs written in the Cartridge version of BASIC.

In this example you will notice that the asterisks under the word "Compiler" are in parentheses. This means that there are differences between the way the statement works under the BASIC interpreter and the way it works under the IBM Personal Computer BASIC Compiler. The IBM Personal Computer BASIC Compiler is an optional software package available

from IBM. If you have the IBM Personal Computer BASIC Compiler, the IBM Personal Computer *BASIC Compiler* manual explains these differences.

Cassette BASIC

The nucleus of BASIC is the Cassette version, which is built into your PCjr in read-only memory. The amount of storage you can use for such things as programs and data depends on how much memory you have in your PCjr. The number of "bytes free" is displayed after you switch on the computer.

The only storage device you can use to save Cassette BASIC information is a cassette tape recorder. You cannot use diskettes with Cassette BASIC.

Some special features you will find in both versions of BASIC are:

- An extended character set of 256 different characters which can be displayed. In addition to the usual letters, numbers, and special symbols, you also have international characters like , , and . You will also find symbols which are commonly used in scientific and mathematical applications, such as Greek letters. There are also a variety of other symbols.

- Graphics capability. You can draw points, lines, and even entire pictures. The screen is *all points addressable* in either low, medium, or high resolution. More information on this can be found in the next chapter.

- Special input/output devices. The PCjr has a speaker which you can use to make sound. Also, BASIC supports a light pen and joystick which help make your programs more interesting as well as more fun.

Cartridge BASIC

Cartridge BASIC is housed on a separate cartridge that can be inserted into either one of the slots located on the front of the PCjr.

Cartridge BASIC, the most extensive form of BASIC available on the PCjr, does everything that Cassette BASIC does, and more. As with the other version, the number of free bytes you have for programs and data is displayed on the screen when you start BASIC.

Key features found in Cartridge BASIC and not found in Cassette BASIC are the following:

- Input/output to diskette in addition to cassette (only if DOS is present). See Appendix B, "BASIC Diskette Input and Output," for special considerations when using diskette files.

- An internal "clock," which keeps track of the date and time, (only if DOS is present).

- Asynchronous communications (RS232) is supported. Refer to Appendix E for details.

- Event trapping. A program can respond to the occurrence of a specific event by "trapping" (automatically branching) to a specific program line. Events include: communications activity, a function key being pressed, the button being pressed on a joystick, play activity, and the light pen being activated.

- Additional screen modes. Six screen modes with 2, 4, or 16 colors available, depending on the screen mode.

- Advanced graphics. Additional statements are CIRCLE, PUT, GET, PAINT, DRAW, VIEW,

WINDOW, PALETTE, and PALETTE USING. These operations make it easier to create more complex graphics.

- Advanced music support. The PLAY statement allows easy usage of the built-in speaker to create musical tones and can support multi-voice tones to your television or external speaker.

- Communications. The TERM statement is used to communicate with other systems.

Notes

Chapter 2. How to Start and Use BASIC

Contents

USING BASIC

Getting BASIC Started

It's easy to start BASIC on the PC*jr*.

If your computer is off:

1. Remove all cartridges.
2. If your PC*jr* has a diskette drive, remove any diskette.
3. Switch on the computer.

The IBM logo screen appears while the computer is checking itself. Then the words "Version C" and the release number are displayed along with the number of free bytes you have available.

If your computer is on:

1. Remove all cartridges.
2. If your PC*jr* has a diskette drive, remove any diskette.
3. Press and hold down the Ctrl and Alt keys, then press the Del key.

The words "Version C" and the release number are displayed along with the number of free bytes available.

To Start Cartridge BASIC When Not Using DOS

If your computer is off:

1. If your PC*jr* has a diskette drive, remove any diskette.

2. Insert the Cartridge BASIC cartridge into either slot.

3. Switch on the computer.

The words "Version J" and the release number are displayed along with the number of free bytes available.

If your computer is on:

1. If your PC*jr* has a diskette drive, remove any diskette.

2. Insert the Cartridge BASIC cartridge into either slot. This causes your system to do a reset.

The words "Version J" and the release number are displayed along with the number of free bytes available.

To Start Cartridge BASIC While Using DOS

1. Insert the Cartridge BASIC cartridge into either slot. This causes your system to do a reset.

2. Enter the command **BASIC** or **BASICA** when the DOS prompt (>) appears.

The words "Version J" and the release number are displayed along with the number of free bytes available.

You can include options with the BASIC command when you start Cartridge BASIC. To learn about these options, see "Options on the BASIC Command" in this chapter.

Returning to DOS from Cartridge BASIC

1. When BASIC prompts you for a command, type:

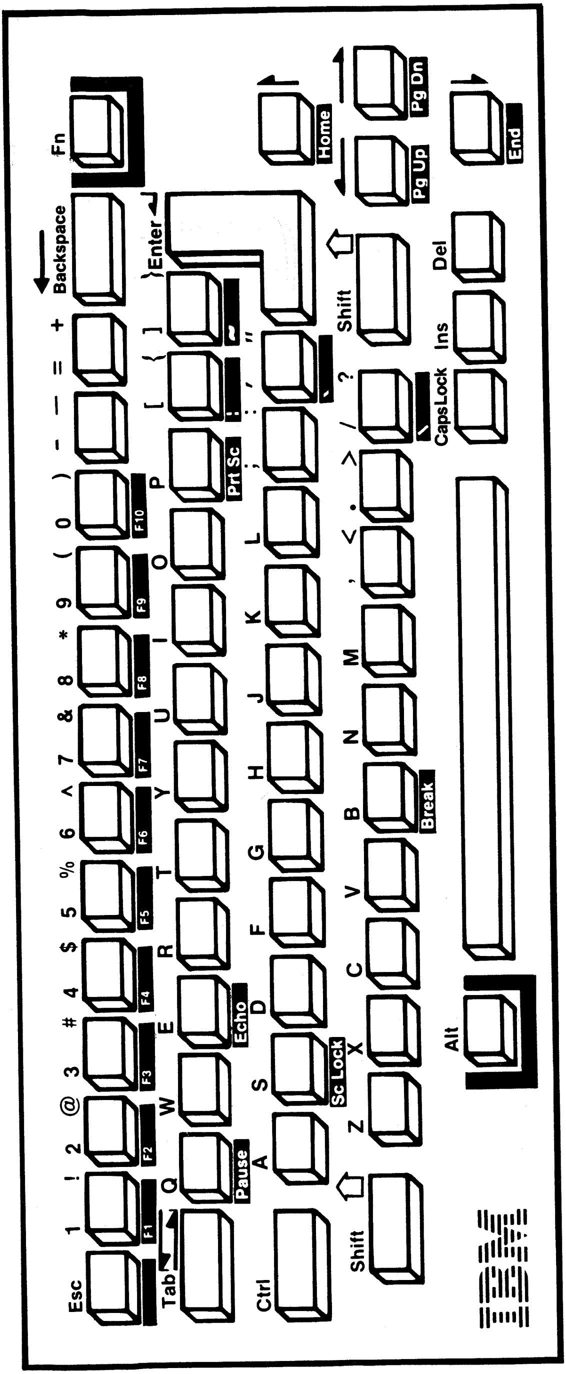
   ```
   SYSTEM
   ```

   then press the Enter key.

2. When you see the DOS prompt (>), DOS is ready for you to give it a command.

For more information, see "SYSTEM Command" in Chapter 4.

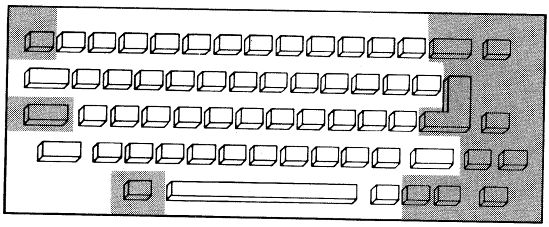
The Keyboard



The keyboard is similar to a typewriter keyboard with some special keys added. These special keys have special functions. Two of these keys are the Alt and Fn keys, which put the keyboard in Alternate mode or Function mode.

All typewriter keys are typematic. This means that each key repeats as long as you hold it down. The typewriter keyboard and the special keys are explained in more detail below.

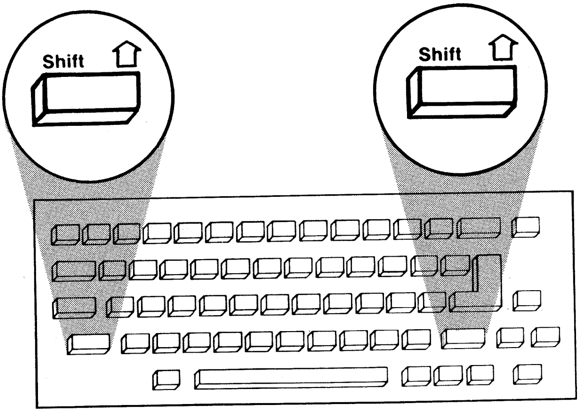
Typewriter Keys



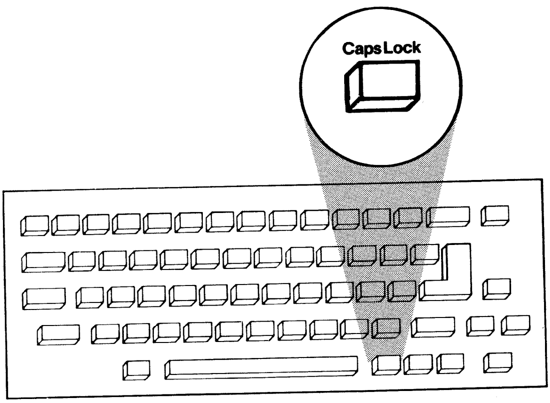
Lowercase Shift

In Lowercase Shift, the typewriter area of the keyboard looks and works much like a standard typewriter. The numbers 0 through 9 are on the top row. The keyboard also has some some special characters not found on a standard typewriter, like [,], <, and >. The letters, numbers, and symbols that the keyboard displays in Lowercase Shift are shown in white above each key.

Uppercase Shift



To put the typewriter keyboard in Uppercase shift, press and hold down one of the Shift keys while you press one of the typewriter keys. In Uppercase shift, the keyboard displays the letters, numbers, and symbols that are shown in black above each key.



Another way of getting uppercase letters is with the Caps Lock key.

After you press this key, you will continue to get capital letters until you press it again. You can get lowercase letters when in Caps Lock state by pressing and holding one of the Shift keys. When you release the Shift key, you'll go back to Caps Lock state.

The Caps Lock key gives you only the uppercase letters. To get the uppershift characters on the numeric or symbol keys, you must use the Shift keys.

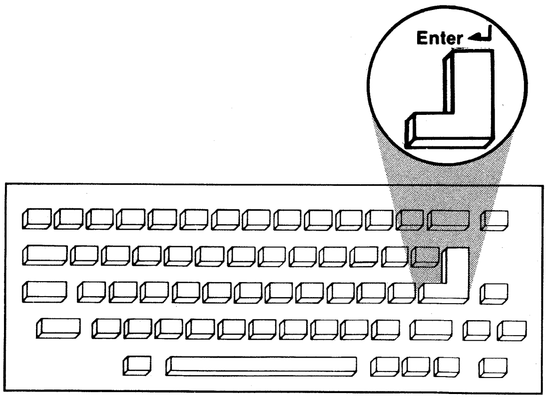
To get out of Caps Lock state, press the Caps Lock key again.

Special Keys

Besides the typewriter keyboard, your PCjr keyboard has special keys. These keys are: Enter, Backspace, Ctrl, Alt, Fn, Esc, Ins, Del, and the four cursor control keys.

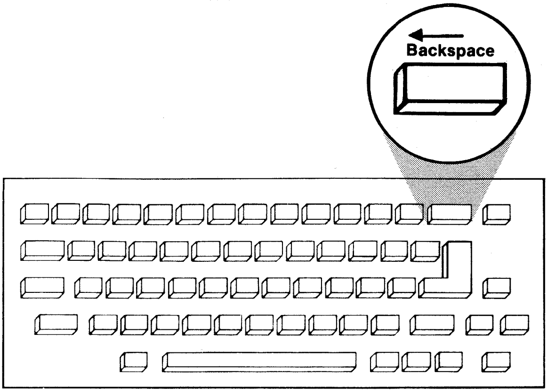
The Ctrl, Alt, and Fn keys put the keyboard into Control, Alternate, and Function modes. These modes are described later in this chapter. The Esc, Ins, Del, and the four cursor control keys are used for editing programs. These keys are described in "The BASIC Program Editor" later in this chapter.

Enter Key



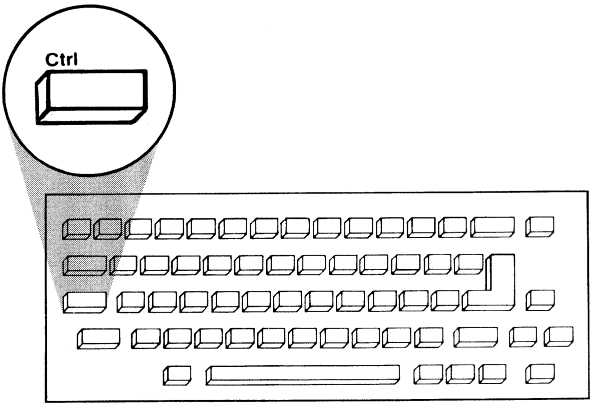
The key shaped like a backwards L is the Enter key. You usually have to press this key to enter information into the computer.

Backspace



The Backspace key behaves somewhat differently from the Backspace key on a typewriter. It not only backspaces, it erases what you've typed as well. If you use the Cursor Left key to backspace, you will not erase what you've typed. Refer to "The BASIC Program Editor" later in this chapter.

Control Mode



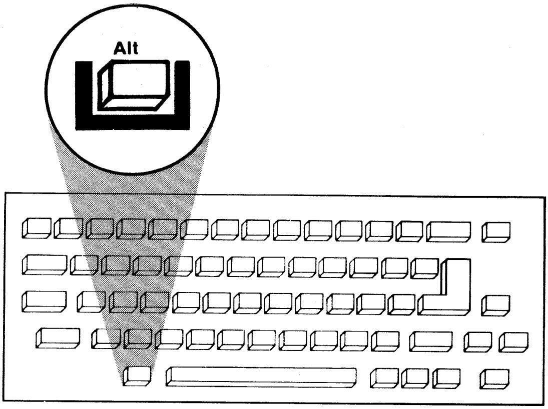
The Ctrl key puts the keyboard in Control mode. Use the Ctrl key like the Shift keys. That is, press and hold the Ctrl key, then press the desired key. Then you can release both keys.

The Ctrl key is used to enter certain codes and characters not otherwise available from the keyboard.

For example, **Ctrl-G** is the *bell* character. When this character is printed, the speaker beeps. The command "Ctrl-G" means that you press and hold the Ctrl key, then press the G key.

You also use the Ctrl key together with other keys to edit programs with the program editor.

Alternate Mode



The Alt key puts the keyboard in Alternate mode. Use the Alt key like the Shift keys. That is, press and hold the Alt key, then press the desired key. Then you can release both keys.

The Alt key lets you enter BASIC statement keywords easily. You can enter an entire BASIC keyword with a single keystroke. The BASIC keyword is typed when the Alt key is held down while one of the alphabetic keys A-Z is pressed.

Keywords associated with each letter are summarized below.

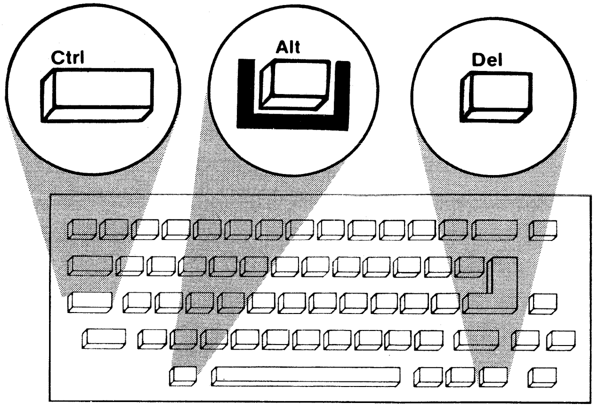
A	AUTO	N	NEXT
B	BSAVE	O	OPEN
C	COLOR	P	PRINT
D	DELETE	Q	(no word)
E	ELSE	R	RUN
F	FOR	S	SCREEN
G	GOTO	T	THEN
H	HEX$	U	USING
I	INPUT	V	VAL
J	(no word)	W	WIDTH
K	KEY	X	XOR
L	LOCATE	Y	(no word)
M	MOTOR	Z	(no word)

The Alt key also lets you display the symbols bordered in blue on your keyboard. To display these symbols, press and hold the Alt key while pressing a key with a blue border.

The Alt key is also used with the number keys to enter characters not found on the keys. This is done by holding down the Alt key and typing the three-digit ASCII code for the character. (See Appendix G, "ASCII Character Codes" for a complete list of ASCII codes.)

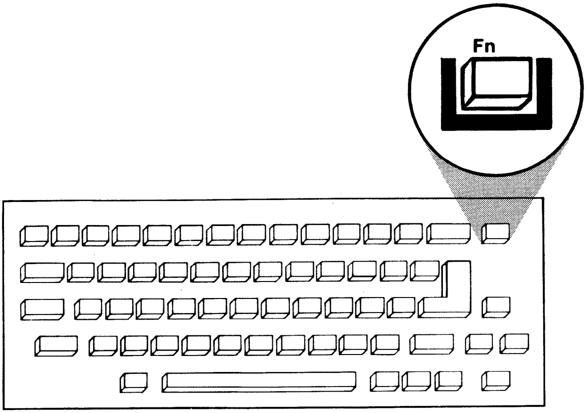
System Reset

The Alt key has a special use when combined with the Ctrl and Del keys.



If the computer power is on, pressing Alt-Ctrl-Del causes a *System Reset*. This is similar to switching the computer from off to on. You must press the Alt and Ctrl keys (in either order) and hold them down, then press the Del key. Then you can release all three keys. Doing a System Reset with these keys is preferable to flipping the power switch off and on again, because the system restarts faster.

Function Mode



The Fn key puts the keyboard in Function mode. To use Function mode, press and release the Fn key, then press a key that has a function assigned to it. The keys bordered in green have already been assigned some frequently used functions. You may change the functions assigned to the numeric keys if you wish.

You do not need to hold down the Fn key while you press another key.

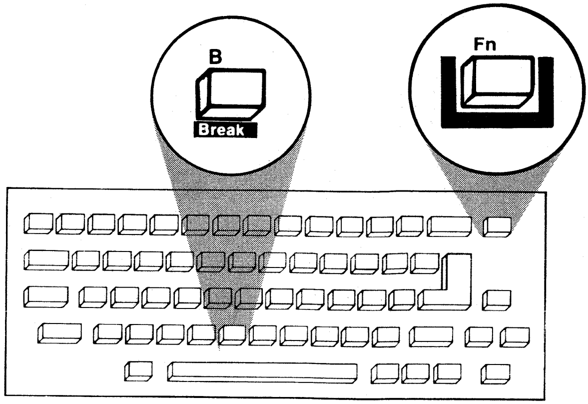
You use the function keys:

- As "soft keys." That is, you can set each key to automatically type any sequence of characters. To change the function assigned to a numeric key or to assign a function to a new key, refer to "KEY Statement" in Chapter 4.

- As program interrupts in Cartridge BASIC, through use of the ON KEY statement. See "ON KEY(n) Statement" in Chapter 4.

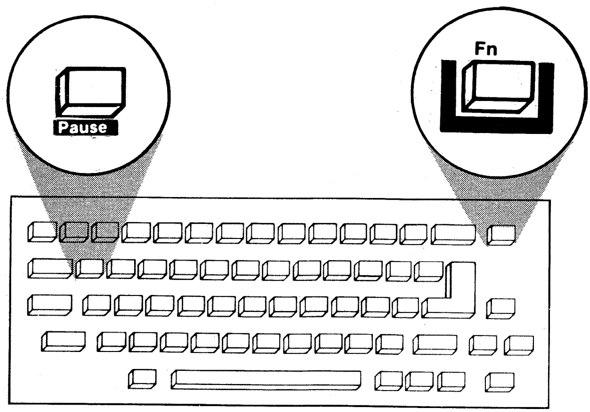
You should be aware of some of the functions assigned to the keys bordered in green. These are described below and in "Special Program Editor Keys" in this chapter.

Break Function



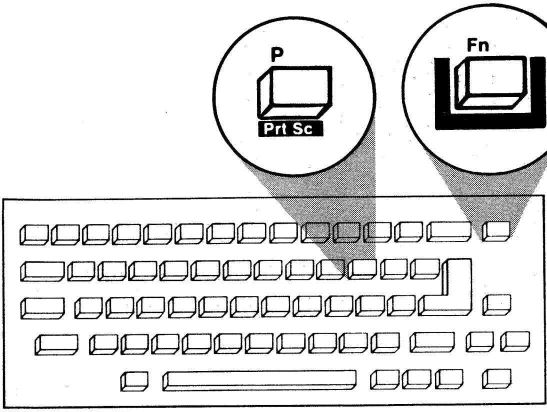
Pressing the Fn key and then the Break key interrupts program execution at the next BASIC instruction and returns to BASIC command level. It is also used to exit AUTO line numbering mode.

Pause Function



Pressing the Fn key and then the Pause key puts the computer in a *pause* state. This can be used to temporarily halt printing or program listing. The pause continues until any key, other than the "shift" keys, the Ins key, or Fn/Break, is pressed.

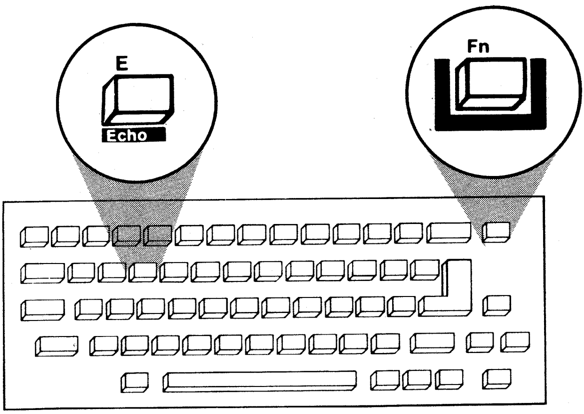
Print Screen Function



Pressing the Fn key and then the PrtSc key causes a copy of what is displayed on the screen to be printed on the printer. Characters which are not recognized by the printer are printed as blanks.

Another way to get a printed copy of your screen is to use the Echo Print function.

Echo Print Function



Pressing the Fn key and then the Echo key serves as an on-off switch allowing text sent to the screen to also be sent to your system printer. Press the Fn key and then press the Echo key to print the text that is on the screen. Press both keys again to stop printing.
In BASIC, use the Fn/PrtSc keys to get a printed copy of the screen.

The BASIC Program Editor

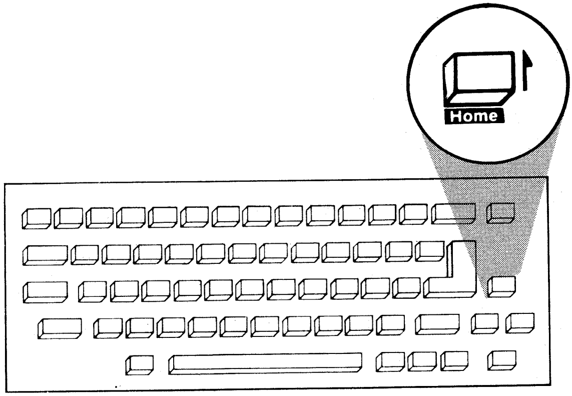
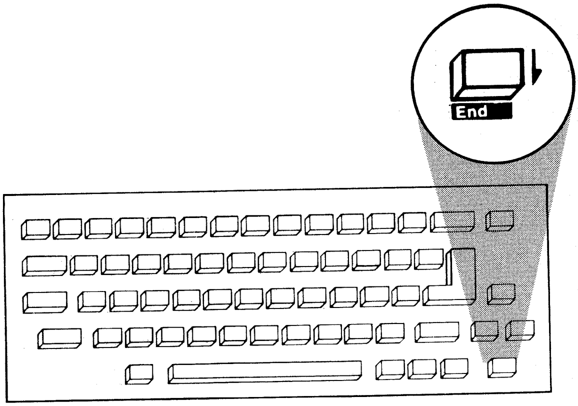
Any line of text typed while BASIC is at the command level is processed by the BASIC program editor. The program editor is a "screen line editor." That is, you can change a line anywhere on the screen, but you can only change one line at a time. The change will only take effect if you press Enter on that line.

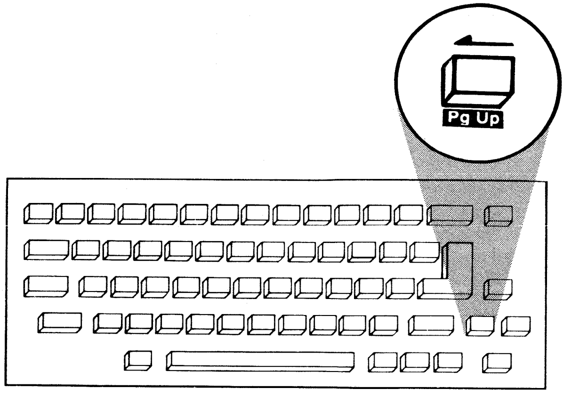
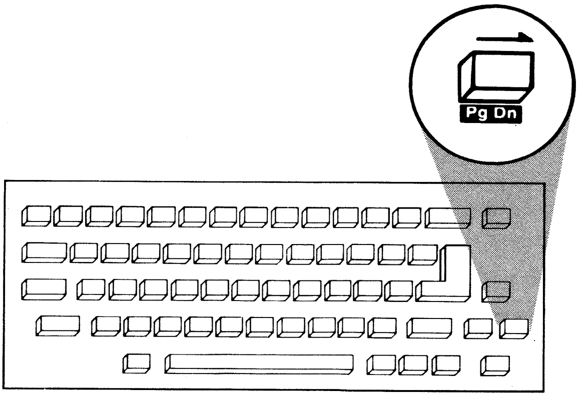
The program editor can save you time during program development. To understand how it works, we suggest you enter a sample program and practice using all of the edit keys. The best way for you to get a "feel" for the editing process is to try editing a few lines while studying the information that follows.

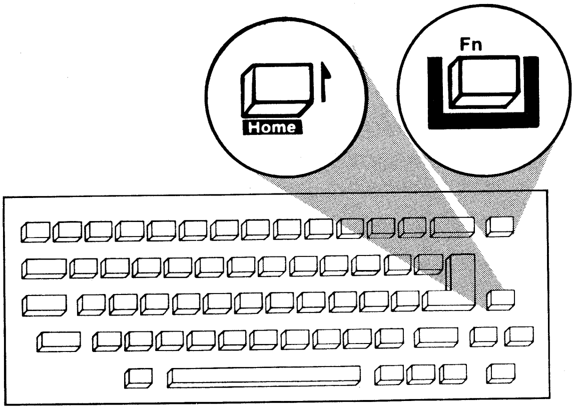
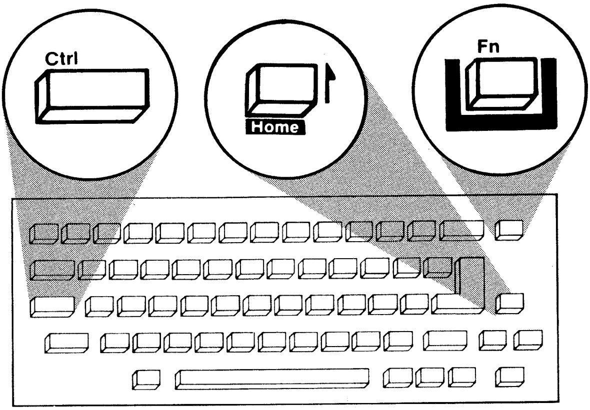
As you type things on your computer, you'll notice a blinking underline or box appearing just to the right of the last character you typed. This line or box is called the *cursor*. It marks the position at which the next character is to be typed, inserted, or deleted.

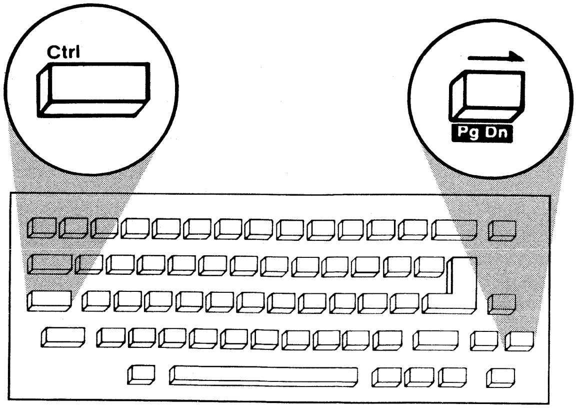
Special Program Editor Keys

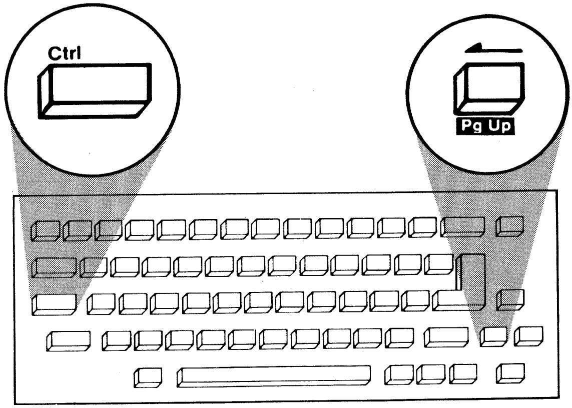
You use the four cursor control keys, the Backspace key, and the Ctrl key to move the cursor to a location on the screen, insert characters, or delete characters. The keys and their functions are shown on the next pages.

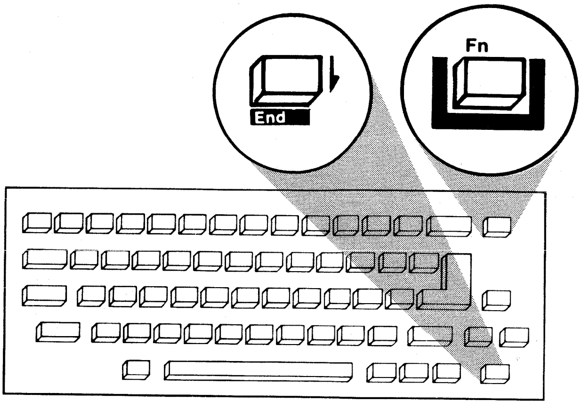
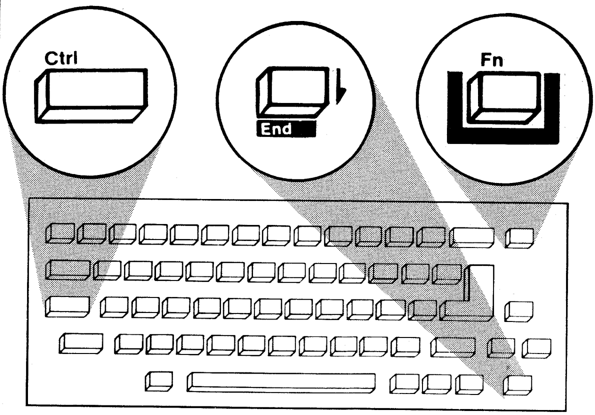
Key(s)	Function
↑ (Cursor Up)	 Moves the cursor one position up.
↓ (Cursor Down)	 Moves the cursor one position down.

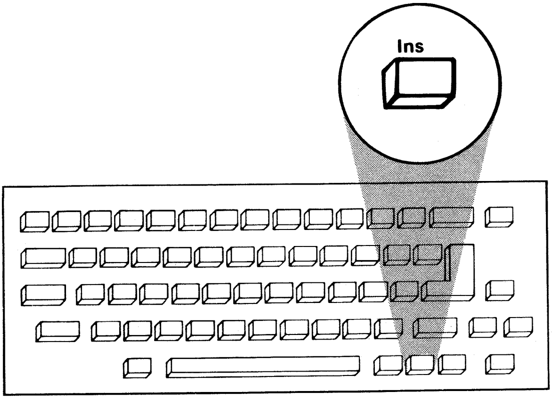
Key(s)	Function
← (Cursor Left)	 Moves the cursor one position left. If the cursor advances beyond the left edge of the screen, the cursor will move to the right side of the screen on the preceding line.
→ (Cursor Right)	 Moves the cursor one position right. If the cursor advances beyond the right edge of the screen, the cursor will move to the left side of the screen on the next line down.

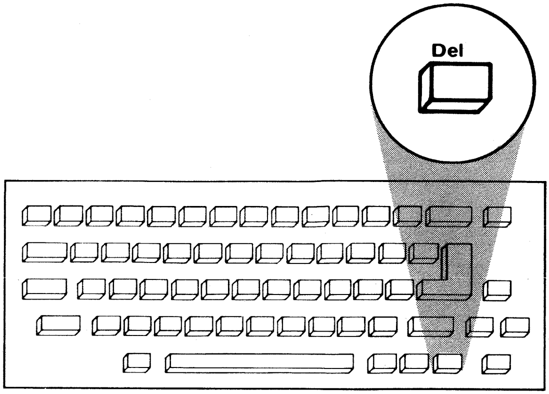
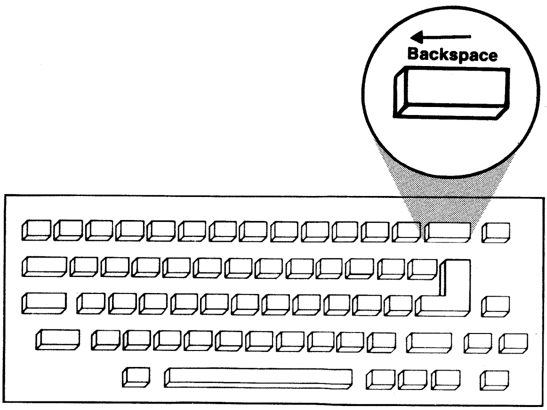
Key(s)	Function
Fn / Home	 Moves the cursor to the upper left-hand corner of the screen.
Ctrl - Fn / Home	 Clears the screen and positions the cursor in the upper left-hand corner of the screen.

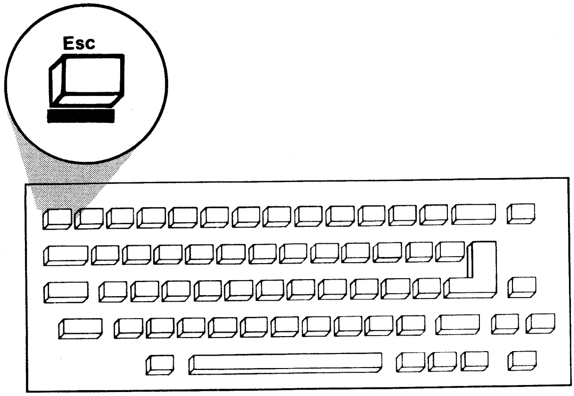
Key(s)	Function
Ctrl - PgDn	 Moves the cursor right to the next *word.* A word is defined as a character or group of characters which begins with a letter or number. Words are separated by blanks or special characters. So, the next word will be the next letter or number to the right of the cursor which follows a blank or special character. For example, suppose the following line is on the screen: **LINE (L1,LOW2)-(MAX,48) ,3 , BF** As you can see, the cursor is presently in the middle of the word L0W2. If we press Next Word (Ctrl-Cursor Right), the cursor will move to the beginning of the next word, which is MAX: **LINE (L1,LOW2)-(MAX,48) ,3 , BF** If we press Next Word again, the cursor will move to the next word, which is the number 48: **LINE (L1,LOW2)-(MAX,48) ,3 , BF**

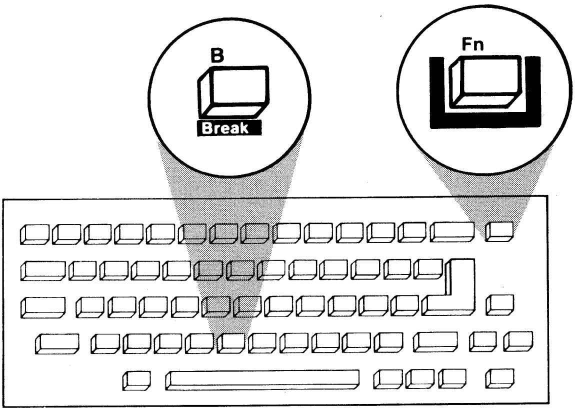
Key(s)	Function
Ctrl - PgUp	 Moves the cursor left to the previous word. The previous word will be the letter or number to the left of the cursor preceded by a blank or special character. For example, suppose we have: **LINE (L1,LOW2)-(MAX,48) ,3 , BF_** If we press Previous Word (Ctrl-Cursor Left), the cursor moves to the beginning of the word BF: **LINE (L1,LOW2)-(MAX,48) ,3 , BF** When we press Previous Word again, the cursor moves to the previous word, which is the number 3: **LINE (L1,LOW2)-(MAX,48) ,3 , BF** And if we press it twice more, the cursor will back up first to the number 48, then to the word MAX: **LINE (L1,LOW2)-(MAX,48) ,3 , BF**

Key(s)	Function
Fn / End	 Moves the cursor to the end of the logical line. Characters typed from this position are added to the end of the line.
Ctrl - Fn / End	 Erases to the end of logical line from the current cursor position. All physical screen lines are erased until the last Enter is found.

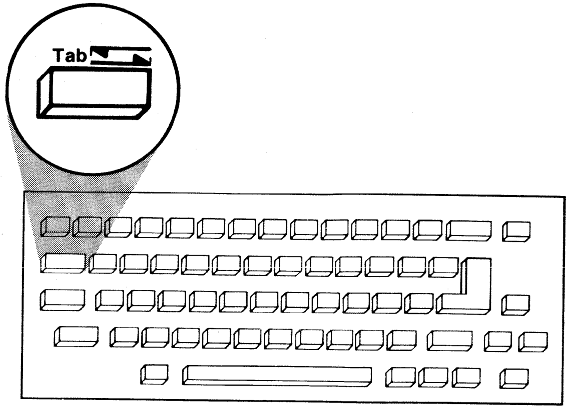
Key(s)	Function
Ins	 Sets Insert mode. If Insert mode is off, then pressing this key will turn it on. If Insert mode is already on, then you will turn it off when you press this key. When you're in Insert mode, the cursor covers the lower half of the character position. When Insert mode is on, characters above and following the cursor move to the right as typed characters are inserted at the current cursor position. After each keystroke, the cursor moves one position to the right. Line folding occurs. That is, as characters advance off the right side of the screen they return on the left of the following line. When Insert mode is off, any characters typed replace existing characters on the line. Besides pressing the Ins key again, Insert mode will also be turned off when you press any of the cursor movement keys or the Enter key.

Key(s)	Function
Del	 Deletes the character at the current cursor position. All characters to the right of the deleted character move one position left to fill in the empty space. Line folding occurs. That is, if a logical line extends beyond one physical line, characters on subsequent lines move left one position to fill in the previous space, and the character in the first column of each subsequent line moves up to the end of the preceding line.
← **Backspace**	 Deletes the last character typed. That is, it deletes the character to the left of the cursor. All characters to the right of the deleted character move left one position to fill in the space. Subsequent characters and lines within the current logical line move up as with the Del key.

<table>
<tr><th>Key(s)</th><th>Function</th></tr>
<tr><td>Esc</td><td>

When pressed anywhere in the line, erases the entire logical line from the screen. The line is not passed to BASIC for processing. If it is a program line, it is not erased from the program in memory.</td></tr>
<tr><td>Fn / Break</td><td>

Returns the computer to command level, without saving any changes that were made to the current line being edited. It does not erase the line from the screen like Esc does.</td></tr>
</table>

Key(s)	Function
Tab	 Moves the cursor to the next tab stop. Tab stops occur every eight character positions; that is, at positions 1, 9, 17, etc. When Insert mode is off, pressing the Tab key moves the cursor over characters until it reaches the next tab stop. For example, suppose you have the following line: **10 REM This is a remark** If you press the Tab key, the cursor will move to the ninth position as shown: **10 REM This is a remark** If you press the Tab key again, the cursor moves to the 17th position on the line: **10 REM This is a remark**

Key(s)	Function
	When Insert mode is on, pressing the Tab key inserts blanks from the current cursor position to the next tab stop. Line folding occurs as explained under **Ins.** For example, suppose we have this line: **10 REM This is a remark** If you press the Ins key and then the Tab key, blanks are inserted up to position 17: **10 REM Th is a remark**

How to Make Corrections on the Current Line

Lines of text typed while BASIC is at the command level are processed by the program editor. You can use any of the keys described in the previous section under "Special Program Editor Keys." BASIC is always at the command level after the prompt **Ok** and until a RUN command is given.

A *logical line* is a string of text which BASIC treats as a unit. You can extend a logical line over more than one physical screen line by simply typing beyond the edge of the screen. The cursor will *wrap* to the next screen line. You can also use a line feed (Ctrl-Enter). Typing a line feed causes subsequent text to be printed on the next screen line without your having to enter all the blanks to move the cursor there. The line is not processed; this only happens when you press Enter.

Note that the line feed actually fills the remainder of the physical screen line with blank characters. A line feed character is not added to the text. These blanks are included in the 255 characters allowed for a BASIC line.

When the Enter key is finally pressed, the entire logical line is passed to BASIC for processing.

Changing Characters

If you made a typing error, you can correct it by moving the cursor to the position where the mistake occurred, and typing the correct letters on top of the wrong ones. Then, move the cursor back to the end of the line using the Cursor Right or Fn/End keys, and continue typing.

For example, suppose we want to load a program called PROGRAM1, and we have typed the following:

```
LOAD "FROG_
```

You accidentally typed **F** instead of **P**. You can fix the problem by pressing Previous Word (Ctrl-Cursor Left) once, until the cursor is under the **F**:

```
LOAD "FROG
```

Then type **P**:

```
LOAD "PROG
```

Then press the Fn/End keys:

```
LOAD "PROG_
```

The error is fixed and you can continue typing:

```
LOAD "PROGRAM1"_
```

Erasing Characters

If you notice you've typed an extra character in the line you're typing, you can erase (delete) it by using the Del key. Use the Cursor Left or other cursor control keys to move the cursor to the character you want to erase. Press the Del key to erase the character. Then use the Cursor Right or Fn/End keys to move the cursor back to the end of the line, and continue typing.

For example, suppose you typed the following:

```
DEELETE_
```

To erase the extra **E**, you press Cursor Left until the cursor is under the extra **E**:

```
DEELETE
```

Then you press the Del key:

```
DELETE
```

Then press the Fn/End keys:

```
DELETE_
```

and continue typing:

```
DELETE 20_
```

If the incorrect character was the character you just typed, use the Backspace key to delete it. Then you can simply continue typing the line as desired.

For example, suppose you have typed the following:

```
DELETT_
```

Simply press the Backspace key:

```
DELET_
```

Then you can continue typing:

```
DELETE 20_
```

Adding Characters

If you see that you've omitted characters in the line you're typing, move the cursor to the position you want to put the new characters. Press the Ins key to get into insert mode. Type the characters you want to add. The characters you type will be inserted at the cursor and the characters above and following it will be pushed to the right. As before, when you're ready to continue typing at the end of the line, use the Cursor Right or Fn/End keys to move the cursor there and just continue typing. Insert mode is turned off when you use either of these keys.

For example, suppose you have typed the following:

```
LIS 10_
```

You forgot the **T** in **LIST**. Press Cursor Left until the cursor is under the space:

```
LIS_10
```

Then you press the Ins key and type the letter **T**:

```
LIST_10
```

Erasing Part of a Line

To break a line at the current cursor position, press the Ctrl-Fn/End keys.

For example, suppose you have the following:

```
10 REM *** garbage garbage garbage
```

You have the cursor positioned under the **g** in the first word **garbage**, so to erase the garbage press the Ctrl-Fn/End keys.

```
10 REM ***
```

Cancelling a Line

To cancel a line that is in the process of being typed, press the Esc key anywhere in the line. You do not have to press Enter. This erases the entire logical line.

For example, suppose you had this line:

```
THIS IS A LINE THAT HAS NO MEANING_
```

Even though the cursor is at the end of the line, the entire line is erased when you press Esc:

Entering or Changing a BASIC Program

Any line of text that you type that begins with a number is considered to be a *program line.*

A BASIC program line always begins with a line number, ends with an Enter, and may contain a maximum of 255 characters, including the Enter. If a line contains more than 255 characters, the extra characters will be truncated when the Enter is pressed. Even though the extra characters still appear on the screen, they are not processed by BASIC.

BASIC keywords and variable names must be in uppercase. However, you may enter them in any combination of uppercase and lowercase. The program editor converts everything to uppercase, except for remarks, DATA statements, and strings enclosed in quotation marks.

BASIC will sometimes change the way you enter something in other ways. For example, suppose you use the question mark (?) instead of the word PRINT in a program line. When you later list the line, the ? will be changed to PRINT with a space after it, since ? is a shorthand way of entering PRINT. This expansion may cause the end of a line to be broken if the line length is close to 255 characters.

> **Warning:** If your line reaches maximum length, the 255th character must be Enter.

Adding a New Line to the Program

Enter a valid line number (range is 0 through 65529) followed by at least one non-blank character, followed by Enter. The line will be saved as part of the BASIC program in storage.

For example, if you enter the following:

```
10 hello Dori
```

This saves the line as line number 10 in the program. Note that **hello Dori** is not a valid BASIC statement. However, you will not get an error if you enter this line. Program lines are *not* checked for proper syntax before being added to the program. That only happens when the program line is actually executed.

If a line already exists with the same line number, then the old line is erased and replaced with the new one.

If you try to add a line to a program when there is no more room in storage, you get an **Out of memory** error and the line is not added.

Replacing or Changing an Existing Program Line

An existing line is changed, as indicated above, when the line number of the line you enter matches the line number of a line already in the program. The old line is replaced with the text of the new one.

For example, if you enter:

```
10 this is a new line 10
```

The previous line 10 (hello Dori) is replaced with this new line 10.

Deleting Program Lines

To delete an existing program line, type the line number alone followed by Enter. For example, if you type:

```
10
```

and press enter, the line 10 is deleted from the program.

Or you may use the DELETE command to delete a group of program lines. Refer to "DELETE Command" in Chapter 4 for details.

Note that if you try to delete a non-existent line, you get an **Undefined line number** error.

Do not use the Esc key to delete program lines. Esc causes the line to be erased from the screen only. If the line exists in the BASIC program, it remains there.

Deleting an Entire Program

To delete the entire program that is currently in memory, enter the NEW command (see "NEW Command" in Chapter 4). NEW is usually used to clear memory before entering a new program.

Changing Lines Anywhere on the Screen

To edit any line on the screen use the cursor control keys (described under "Special Program Editor Keys") to move the cursor on the screen to the place requiring the change. Then you can use any or all of the procedures described previously to change, delete, or add characters to the line.

If you want to modify program lines that do not happen to be displayed at the moment, you can use the LIST command to display them. List the line or range of lines to be edited (see "LIST Command" in Chapter 4).

Position the cursor at the line to be edited and change the line using the procedures already described. Press Enter to store the modified line in the program. You can also use the EDIT command to display the line you want. Refer to "EDIT Command" in Chapter 4.

For example, you can duplicate a line in the program this way: move the cursor to the line to be duplicated. Then, change the line number to the new line number by just typing over the numbers. When you press Enter, both the old line and the new line are in the program.

Or, you can change the line number of a program line by duplicating the line as described above, then deleting the old line.

A program line is never actually changed within the BASIC program until you press Enter. Therefore, when several lines need changes, it may be easier to move around the screen making corrections to several lines at once. When you've made all the changes, move the cursor to the beginning of each changed line and press Enter. By so doing, you store each changed line in the program.

You do not have to move the cursor to the end of the logical line before pressing Enter. The program editor knows where each logical line ends and it processes the whole line even if the Enter is pressed at the beginning of the line.

> **Note:** Use of the AUTO command can be very helpful when you are entering your program. However, you must exit AUTO mode by pressing the Fn/Break keys before changing any lines other than the current one.

Remember, changes made using these procedures only change the program in memory. To save the program

with the new changes permanently, you should use the SAVE command (see “SAVE Command” in Chapter 4) before entering a NEW command or leaving BASIC.

Syntax Errors

When a syntax error is discovered while a program is running, BASIC displays the line that caused the error so you can correct it. For example:

```
Ok
10 A = 2$12
RUN
Syntax error in 10
Ok
10 A = 2$12
```

The program editor has displayed the line in error and positioned the cursor under the digit 1. You can move the cursor to the dollar sign ($) and change it to a plus sign (+), then press Enter. The corrected line is now stored in the program.

When you edit a line and store it back in the program while the program is interrupted (as in this example) the following happens:

- All variables and arrays are lost. That is, they are reset to zero or null.
- Any files that were open are closed.
- You cannot use CONT to continue the program.

If you want to examine the contents of some variable before making the change, press the Fn/Break keys to return to command level. The variables are preserved since no program line is changed. After you check everything you need to, edit the line and rerun the program.

Modes of Operation

Once BASIC is started, it displays the prompt **Ok**. **Ok** means BASIC is ready for you to tell it what to do. This state is known as *command level*. At this point, you may talk to BASIC in either of two modes: the *direct mode* or the *indirect mode*.

Direct Mode

Direct mode means you are telling BASIC to perform your request immediately after the request is entered. You tell BASIC to do this by *not* preceding the statement or command with a line number. You can display results of arithmetic and logical operations immediately or store them for later use, but the instructions themselves are not saved after they are executed. This mode is useful for debugging as well as for quick arithmetic operations that do not require a complete program. For example:

```
Ok
PRINT 20+2
 22
Ok
```

Indirect Mode

You enter programs using indirect mode. To tell BASIC that the line you are entering is part of a program, begin the line with a *line number*. The line is then stored as part of the program in memory. The program in storage begins when the RUN command is entered. For example:

```
Ok
1 PRINT 20+2
RUN
 22
Ok
```

Running a BASIC Program

Two steps are involved in running a BASIC program stored on a diskette, cassette or cartridge.

The first step is getting a copy of the program transferred into the computer's memory. This is called *loading* the program and is done with the LOAD command.

The second step is the actual performance of the program's instructions. This is called *running* the program and is done with the RUN command.

Running a Program on Diskette

If you have the DOS Supplemental Programs diskette that comes with DOS, let's go through the sequence of loading and running a program. You will use the SAMPLES program on this diskette.

You must use the Cartridge BASIC cartridge if you have a DOS system with a diskette drive.

Running the SAMPLES Program

1. Make sure DOS is ready and **A>** is displayed on the screen.

2. Insert the DOS diskette if it is not already in the diskette drive.

3. Insert the Cartridge BASIC cartridge in either slot. This causes your system to reset.

4. When the DOS prompt appears (>), type:

   ```
   basic
   ```

   and press the Enter key.

   You see the BASIC prompt, **Ok**.

5. Now remove the DOS diskette and insert the DOS Supplemental Programs diskette.

6. Type:

   ```
   load "samples
   ```

   and press the Enter key.

7. When you see **Ok**, type:

```
run
```

and press the Enter key.

When the following screen is displayed press the space bar.

```
                    IBM

              Personal Computer

              SAMPLES

              Version 2.00

     (C) Copyright IBM Corp 1982, 1983

          Press space bar to continue
```

8. Next you see the *menu* screen. You select the item you want from a fixed number of choices, as you would from a restaurant menu.

```
SAMPLE PROGRAMS

A — MUSIC    (64k)
B — ART      (48k—Color/Graphics)
C — MORTGAGE (64k)
D — CIRCLE   (BASICA—Color/Graphics)
E — DONKEY   (BASICA—Color/Graphics)
F — PIECHART (BASICA—Color/Graphics)
G — BALL     (BASICA—Color/Graphics)
H — COLORBAR (48k)
I — SPACE    (BASICA—Color/Graphics)
ESC KEY — EXIT

ENTER LETTER OF PROGRAM

NOTE: All of the above programs
      require 64k if using BASICA
```

With your PCjr, you have at least 64K of memory and either a color television or a Color/Graphics Monitor, so you can choose any of the menu items if you are using Cartridge BASIC.

Let's try choice H—COLORBAR. Type:

```
h
```

You do not need to press the Enter key. Follow the directions on the screen to see the computer display the different colors.

9. Return to the menu and select any of the other items you want.

10. When you have tried any or all of these programs, press the Esc key. You see the BASIC prompt, **Ok**. Type:

```
system
```

and press the Enter key.

This gets you back to DOS.

Running the COMM Program

A sample telecommunications program is also provided on the DOS Supplemental Programs diskette. This program lets you establish an asynchronous communications link between your PCjr and another PCjr, an IBM Series/1 computer, or two communications network services.

This means that your computer can "talk" to another computer or be part of a network service. Using a network service is like being on a telephone "party line."

The COMM program will work only if you have the necessary equipment, and subscriptions. If you need help using external devices, consult your dealer.

You could also use the COMM program as a model for writing your own telecommunications program.

Let's look at the COMM program menu. (You can do this even if you don't plan to communicate with another computer.) Follow these steps:

1. Make sure Cartridge BASIC is running and **Ok** is displayed.

2. Insert the DOS Supplemental Programs diskette in drive A if it is not already there.

3. Type:

   ```
   load "comm
   ```

   and press the Enter key.

4. Now type:

   ```
   run
   ```

   and press the Enter key.

5. The sample communications program menu is displayed.

```
                 COMMUNICATIONS MENU


        Choose one of the following:

                 1 Description of program
                 2 Dow Jones/News Retrieval
                 3 IBM Personal Computer
                 4 Series/1
                 5 THE SOURCE
                 6 Other service
                 7 End program

        Choice
```


6. You can use options 1 or 7 now, even if you are not ready to establish communications.

7. Each choice (except 7) will lead you to another menu screen.

   When you're through reading the information, press the Fn/1 keys. The main menu is displayed again.

8. Type:

   7

   and press the Enter key.

   You are back in Cartridge BASIC.

COMM Program Choices

Here's a short description of the COMM program choices.

1. **Description of program**

   This choice displays a screen that describes the COMM program.

2. **Dow Jones/News Retrieval**

   This choice lets you dial in to the Dow Jones/News service.

   You must have a Dow Jones/News service subscription, as well as the communications equipment, to run this choice.

3. **PCjr**

   This choice lets your PCjr communicate with another PCjr.

   Can also be used to let your PCjr communicate with another PCjr.

4. **Series/1**

   This choice lets your PCjr communicate with an IBM Series/1 computer, running either Realtime Programming System (RPS) V5.1 or Event Driven Executive (EDX) V3.0.

5. **THE SOURCE**

   This choice connects your computer to THE SOURCE service.

   To use this choice, you need to get a subscription to THE SOURCE and purchase the necessary communication equipment.

6. **Other service**

   This choice lets you describe the kind of communications your PCjr will set up. You do this if the choices that were made in the COMM program are not correct for your case. Then you can start the communications session using the characteristics you've described.

7. **End program**

   This option ends the COMM program and takes you back into BASIC. You then see the BASIC prompt, **Ok**.

Running a BASIC Program on Another Diskette

For this example, let's assume that the Cartridge BASIC program you want to run is called BOWLING and it is not on your DOS diskette.

1. Make sure DOS is ready and **A>** is displayed.

2. Insert the DOS diskette if it is not already in the diskette drive.

3. Insert the Cartridge BASIC cartridge int either slot. This causes your system to reset.

4. When the DOS prompt appears (>), type:

   ```
   basic
   ```

   press the Enter key.

   You will see the BASIC prompt, **Ok**.

5. Now remove the DOS diskette and insert the diskette that contains the BOWLING file.

6. Type:

   ```
   load "bowling
   ```

   press the Enter key.

USING BASIC

Note: If you do not supply an extension in the command, BASIC will look for a file with the extension .BAS. In this case, BASIC will look for a file named BOWLING.BAS.

7. When you see the BASIC prompt again, type:

```
run
```

press the Enter key.

8. Now the BOWLING program will perform the instructions in the program.

Running a Program on Cassette

Let's assume that you have a program called BUDGET on a cassette, and you have brought your machine up in Cassette BASIC.

1. After you see the BASIC prompt **Ok**, insert the cassette and type:

```
load "budget
```

press the Enter key.

Note: BASIC looks for a BASIC program file on the cassette tape. It skips over any data files.

2. When you see the BASIC prompt again, type:

```
run
```

press the Enter key.

3. Now the BUDGET program will perform the instructions in the program.

There's another way to run a program you have on cassette.

1. Insert the cassette that has the program you want to run.

2. Switch on the computer, if it is not already on, or perform a system reset by pressing Alt-Ctrl-Del.

3. Press the Ctrl-Esc keys. This will cause the first program on the cassette to be loaded and run.

When you use the Ctrl-Esc keys, the program that is loaded and run is the first one on the cassette. If you are already using the cassette when you press the Ctrl-Esc keys, the next program on the cassette is loaded and run.

Options on the BASIC Command

You can include options on the **BASIC** command when you start Cartridge BASIC. These options specify the amount of storage BASIC uses to hold programs and data, and for buffer areas. You can also ask BASIC to immediately load and run a program.

> **Note:** You must have DOS to use options on the BASIC command.

These options are not required—BASIC works just fine without them. So if you're new to BASIC, you may wish to skip over this section and go on to the next section. Then you can refer to this section when you become more familiar with BASIC and its capabilities.

The complete format of the **BASIC** command is:

BASIC[A] [*filespec*] [<*stdin*] [>] [>*stdout*] [/F:*files*] [/S:*bsize*] [/C:*combuffer*] [/M:[*max workspace*] [, *max blocksize*]]

filespec This is the file specification of a program to be loaded and run immediately. It must be a character string constant, but it should *not* be enclosed in quotation marks. *filespec* is expanded to allow the specification of a *path*. It should conform to the rules for specifying files described under "Naming Files" in Chapter 3. A default extension of .BAS is used if none is supplied and the length of the filename is eight characters or less. If you include *filespec*, BASIC proceeds as if a RUN *filespec* command were the first thing you entered once BASIC was ready. Note that when you specify *filespec*, the BASIC heading with the copyright notices is not displayed.

<stdin A BASIC program normally receives its input from the keyboard (standard input device). Using *<stdin* allows BASIC to receive input from the file you specify. When you use *<stdin*, you must position it before any switches. Refer to "Redirection of Standard Input and Standard Output" in this chapter for more information.

>stdout A BASIC program normally writes its output to the screen (standard output device). Using *>stdout* allows BASIC to write output to the file or device you specify. When you use *>stdout*, you must position it before any switches. Refer to "Redirection of Standard Input and Standard Output," later in this chapter, for more information.

Options beginning with a slash (/) are called switches. A *switch* is a means used to specify parameters. The following options on the BASIC command line are switches.

/F:*files* This sets the maximum number of files that may be open at any one time during the running of a BASIC program. Each file requires 188 bytes of memory for the file control block, plus the buffer size specified in the **/S:** switch. If the **/F:** switch is omitted, the number of files defaults to three. The maximum value for BASIC is 15. The actual number of files that may be open simultaneously depends upon the value of the FILES= parameter in the DOS configuration file, CONFIG.SYS. The default, if not specified in CONFIG.SYS, is FILES=8. BASIC uses four files by default, leaving four files for BASIC file I/O. Therefore,

/F:4 is the maximum value that you can give when FILES=8 and you want to be able to have all files open at the same time.

/S:*bsize* This switch sets the buffer size for use with random files. The record length parameter on the OPEN statement may not exceed this value. The default buffer size is 128 bytes. The maximum value you may enter is 32767 bytes.

/C:*combuffer*
This sets the size of the buffer for receiving data when using the communications. The buffer for transmitting data with communications is always allocated 128 bytes. The maximum value you may enter for the **/C:** switch is 32767 bytes. If the **/C:** switch is omitted, 256 bytes are allocated for the receive buffer. If you have a high-speed line, we suggest you use **/C:1024**. If you also have the IBM internal modem option on your system, both receive buffers are set to the size specified by this switch. You may disable RS232 support by using a value of zero (**/C:0**), in which case no buffer space will be reserved for communications.

/M:*max workspace*
This option sets the maximum number of bytes that may be used as BASIC workspace. BASIC is able to use a maximum of 64K bytes of memory, so the highest value you may set is 64K (hex FFFF). You can use this option to reserve space for machine language subroutines or for special data storage. You may wish to refer to "Memory Map" in Appendix I for more detailed information on how BASIC

uses memory. If the **/M:** switch is omitted, all available memory to a maximum of 64K bytes is used.

max blocksize

If you intend to load programs above the BASIC workspace, then use the optional parameter *max blocksize* with the **/M:** switch to reserve space for the workspace and your programs. The parameter *max blocksize* must be in paragraphs, which are byte multiples of 16. When this parameter is omitted, 4096 (&H1000) is assumed. This allocates 65536 bytes for BASIC's DATA and STACK segment (because 4096 x 16 = 65536). If you want to allocate 65536 bytes for the BASIC workspace and 512 bytes for machine language subroutines, use **/M:,4112**. This gives you 4096 paragraphs for BASIC and 16 paragraphs for your routines. Designating **/M:,2048** means that BASIC will allocate and use 32768 bytes (2048 x 16 = 32768) maximum for the BASIC workspace. Designating **/M:32000,2048** means that BASIC will allocate 32768 bytes maximum (2048 x 16 = 32768), but use only the lower 32000. This leaves 768 bytes free for program space.

Note: The options *files, max workspace, max blocksize, bsize*, and *combuffer* are all numbers that may be either decimal, octal (preceded by &O), or hexadecimal (preceded by &H).

Examples of how to use the BASIC command:

BASIC PAYROLL.BAS

This starts Cartridge BASIC so that it uses the defaults as described – all memory and three files. The program PAYROLL.BAS is loaded and run.

BASIC INVEN/F:6

Here we start Cartridge BASIC to use all memory and six files, and load and run INVEN.BAS. Remember, **.BAS** is the default extension.

BASIC /M:32768

This command starts Cartridge BASIC so the maximum workspace size is 32768 bytes. That is, BASIC will use only 32K bytes of memory. No more than three files will be used at one time.

BASIC CHKWRR.TST/F:2/M:&H9000

This command sets the maximum workspace size to hex 9000. This means Cartridge BASIC can use up to 36K bytes of memory. Also, file control blocks are set up for two files, and the program CHKWRR.TST on the diskette is loaded and run.

Redirection of Standard Input and Standard Output

You can redirect your BASIC input and output. Standard input, normally read from the keyboard, can be redirected to any file you specify on the BASIC command line. Standard output, normally written to the screen, can be redirected to any file or device you specify on the BASIC command line.

Note that this requires the use of DOS 2.10.

BASIC *filespec* [<*stdin*] [>] [>*stdout*]

Examples:

1. In the following example, data read by INPUT, INPUT$, INKEY$, and LINE INPUT will continue to come from the keyboard. Data written by PRINT will go into the DATA.OUT file.

   ```
   BASIC MYPROG >DATA.OUT
   ```

2. In this example, data read by INPUT, INPUT$, INKEY$, and LINE INPUT will come from the DATA.IN file. Data written by PRINT will continue to go to the screen.

   ```
   BASIC MYPROG <DATA.IN
   ```

3. In the next example, data read by INPUT, INPUT$, INKEY$, and LINE INPUT will come from the MYINPUT.DAT file and data written by PRINT will go into the MYOUTPUT.DAT file.

   ```
   BASIC MYPROG <MYINPUT.DAT >MYOUTPUT.DAT
   ```

4. In the last example, data read by INPUT, INPUT$, INKEY$, and LINE INPUT will come from the \SALES\JOHN\TRANS file. Data output written by PRINT will be *added* to the \SALES\SALES.DAT file.

```
BASIC MYPROG <\SALES\JOHN\TRANS. >>\SALES\SALES.DA
```

Notes:

1. When redirected, all INPUT, INPUT$, INKEY$, and LINE INPUT statements read from the specified input file, instead of from the keyboard.

2. When redirected, all PRINT statements write to the specified output file or device, instead of to the screen.

3. Error messages still go to standard output (the screen). All files are closed, the program ends, and control returns to DOS.

4. INKEY$, INPUT$, and file input from "KYBD:" still read from the keyboard.

5. File output to "SCRN:" still goes to the screen.

6. BASIC continues to trap keys when an ON KEY(*n*) statement is specified.

7. The Fn/Echo keys do not give a printed copy of the screen when standard output is redirected.

8. The Fn/Break keys go to standard output, close all files, and return control to DOS.

Chapter 3. General Information about Programming in BASIC

Contents

GENERAL INFO

Line Format

Program lines in a BASIC program have the following format:

```
nnnnn BASIC statement[:BASIC statement...][' comment ]
```

and they end with Enter. This format is explained in more detail below.

Line Numbers

"nnnnn" shows the line number, which can be from one to five digits. Every BASIC program line begins with a line number. Line numbers are used to show the order in which the program lines are stored in memory and also as reference points for branching and editing. Line numbers must be in the range 0 to 65529. A period (.) may be used in LIST, AUTO, DELETE, and EDIT commands to refer to the current line.

BASIC Statements

A BASIC statement is either *executable* or *nonexecutable*. Executable statements are program instructions that tell BASIC what to do next while running a program. For example, **PRINT X** is an executable statement. Nonexecutable statements, such as DATA or REM, do not cause any program action when BASIC sees them. All the BASIC statements are explained in detail in the next chapter.

You may, if you wish, have more than one BASIC statement on a line, but each statement on a line must be separated from the last one by a colon, and the total number of characters must not exceed 255.

For example:

```
Ok
10 FOR I=1 TO 5: PRINT I: NEXT
RUN
 1
 2
 3
 4
 5
Ok
```

Comments

Comments may be added to the end of a line using the ' (single end quote) to separate the comment from the rest of the line.

Character Set

The BASIC character set consists of alphabetic characters, numeric characters and special characters. These are the characters which BASIC recognizes.

The alphabetic characters in BASIC are the uppercase and lowercase letters of the alphabet. The numeric characters are the digits 0 through 9.

The following special characters have specific meanings in BASIC:

Character	Name
	blank
=	equal sign or assignment symbol
+	plus sign or concatenation symbol
-	minus sign
*	asterisk or multiplication symbol
/	slash or division symbol
\	backslash or integer division symbol
^	caret or exponentiation symbol
(	left parenthesis
)	right parenthesis
%	percent sign or integer type declaration character
#	number (or pound) sign, or double-precision type declaration character
$	dollar sign or string type declaration character
!	exclamation point or single-precision type declaration character
&	ampersand
,	comma
.	period or decimal point
'	single quotation mark (apostrophe), or remark delimiter
;	semicolon
:	colon or statement separator
?	question mark (PRINT abbreviation)
<	less than
>	greater than
"	double quotation mark or string delimiter
_	underline

Many characters can be printed or displayed even though they have no particular meaning to BASIC. See Appendix G, "ASCII Character Codes," for a complete list of these characters.

Reserved Words

Certain words have special meaning to BASIC. These words are called *reserved words*. Reserved words include all BASIC commands, statements, function names, and operator names. Reserved words cannot be used as variable names.

You should always separate reserved words from data or other parts of a BASIC statement by using spaces or other special characters as allowed by the syntax. That is, the reserved words must be appropriately *delimited* so that BASIC will recognize them.

The following section lists all of the reserved words in BASIC.

ABS
AND
ASC
ATN
AUTO
BEEP
BLOAD
BSAVE
CALL
CDBL
CHAIN
CHDIR
CHR$
CINT
CIRCLE
CLEAR
CLOSE
CLS
COLOR
COM
COMMON
CONT
COS
CSNG
CSRLIN
CVD
CVI
CVS
DATA
DATE$
DEF
DEFDBL
DEFINT
DEFSNG
DEFSTR
DELETE
DIM
DRAW
EDIT
ELSE
END
ENVIRON
ENVIRON$
EOF
EQV
ERASE
ERDEV
ERDEV$

ERL
ERR
ERROR
EXP
FIELD
FILES
FIX
FN*xxxxxxxx*
FOR
FRE
GET
GOSUB
GOTO
HEX$
IF
IMP
INKEY$
INP
INPUT
INPUT#
INPUT$
INSTR
INT
INTER$
IOCTL
IOCTL$
KEY
KILL
LEFT$
LEN
LET
LINE
LIST
LLIST
LOAD
LOC
LOCATE
LOF
LOG
LPOS
LPRINT
LSET
MERGE
MID$
MKDIR
MKD$
MKI$
MKS$
MOD
MOTOR
NAME
NEW
NEXT
NOISE
NOT
OCT$
OFF
ON
OPEN
OPTION
OR
OUT
PAINT
PALETTE
PCOPY
PEEK
PEN
PLAY
PMAP
POINT
POKE
POS
PRESET
PRINT
PRINT#
PSET
PUT
RANDOMIZE
READ
REM
RENUM
RESET
RESTORE
RESUME

RETURN
RIGHT$
RMDIR
RND
RSET
RUN
SAVE
SCREEN
SGN
SHELL
SIN
SOUND
SPACE$
SPC(
SQR
STEP
STICK
STOP
STR$
STRIG
STRING$
SWAP
SYSTEM
TAB(
TAN
TERM
THEN
TIME$
TIMER
TO
TROFF
TRON
USING
USR
VAL
VARPTR
VARPTR$
VIEW
WAIT
WEND
WHILE
WIDTH
WINDOW
WRITE
WRITE#
XOR

Constants

Constants are the actual values BASIC uses during execution. There are two types of constants: string (or character) constants, and numeric constants.

A string constant is a sequence of up to 255 characters enclosed in double quotation marks. Examples of string constants:

```
"HELLO"
"$25,000.00"
"Number of Employees"
```

There are a few cases where BASIC knows that a particular thing must be a string constant, and the quotation marks are not required. These cases are noted where appropriate in this book. Also, if you start a string with a quotation mark, but forget to add the end quotation mark, BASIC is smart enough to know what the problem is and automatically assumes that a quotation mark is at the end of the line. However, this produces correct results only when the string is the last thing on the line.

Numeric constants are positive or negative numbers. A plus sign (+) is optional on a positive number. Numeric constants in BASIC cannot contain commas. There are five ways to indicate numeric constants:

Integer Whole numbers between -32768 and +32767, inclusive. Integer constants do not have decimal points.

Fixed point Positive or negative real numbers, that is, numbers that contain decimal points.

Floating point

Positive or negative numbers represented in exponential form (similar to scientific notation). A floating point constant consists of an optionally signed integer or fixed point number (the mantissa) followed by the letter E and an optionally signed integer (the exponent). Double-precision floating point constants use the letter D instead of E. For more information, see the next section, "Numeric Precision." The E (or D) means "times ten to the power of." For example,

```
23E-2
```

Here, 23 is the mantissa, and -2 is the exponent. This number could be read as "23 times 10 to the negative two power."You could write it as 0.23 in regular fixed-point notation.

More examples:

```
235.988E-7
```

is a single-precision number that is equivalent to .0000235988.

```
2359D6
```

is a double-precision number that is equivalent to 2359000000.

Remember, when you read floating point notation: the E indicates single-precision calculation and the D indicates double-precision calculation.

You can represent any number from 2.9E-39 to 1.7E+38 (positive or negative) as a floating point constant.

Hex

Hexadecimal numbers with up to four digits, with a prefix of &H. Hexadecimal digits are the numbers 0 through 9, A, B, C, D, E, and F. Examples:

```
&H76
&H32F
```

Octal

Octal numbers with up to six digits, with the prefix &O or just &. Octal digits are 0 through 7. Examples:

```
&O347
&1234
```

Numeric Precision

Numbers may be stored internally as either integer, single-precision, or double-precision numbers. Constants entered in integer, hex, or octal format are stored in two bytes of memory and are interpreted as integers or whole numbers. With double-precision calculation, the numbers are stored with 17 digits of precision and printed with up to 16 digits. With single-precision calculation, seven digits are stored and up to seven digits are printed, although only six digits may be accurate. Seven digits are printed in single precision because any intermediary processing uses all seven digits. To ensure the accuracy of final results, enter all seven digits *before* the final results when you do interactive processing.

A single-precision constant is any numeric constant that is written with:

- seven or fewer digits, or
- exponential form using E, or
- a trailing exclamation point (!)

A double-precision constant is any numeric constant that is written with:

- eight or more digits, or
- exponential form using D, or
- a trailing number sign (#)

The following table summarizes the precision and range of integers, single-precision numbers, and double-precision numbers:

TYPE	RANGE	ACCURACY
Integer	-32768 to 32767	Perfect
Single-precision floating point	10E-38 to 10+38	6 decimal digits
Double-precision floating point	10D-38 to 10D+38	16 decimal digits

Examples of single- and double-precision constants:

Single-Precision	**Double-Precision**
46.8	345692811
-1.09E-06	-1.09432D-06
3489.0	3489.0#
22.5!	7654321.1234

Variables

Variables are names used to represent values that are used in a BASIC program. As with constants, there are two types of variables: numeric and string. A numeric variable always has a value that is a number. A string variable may only have a character string value.

The length of a string variable is not fixed, but may be anywhere from 0 (zero) to 255 characters. The length of the string value you assign to it will determine the length of the variable.

You may set the value of a variable to a constant, or you can set its value as the result of calculations or various data input statements in the program. In either case, the variable type (string or numeric) must match the type of data that is being assigned to it.

If you use a numeric variable before you assign a value to it, its value is assumed to be zero. String variables are initially assumed to be null; that is, they have no characters in them and have a length of zero.

How to Name a Variable

BASIC variable names may be any length. If the name is longer than 40 characters, however, only the first 40 characters are significant.

The characters allowed in a variable name are letters and numbers, and the decimal point. The first character must be a letter. Special characters which identify the type of variable are also allowed as the last character of the name. For more information about types, see the next section, "How to Declare Variable Types."

A variable name may not be a reserved word, but may contain imbedded reserved words. (Refer to "Reserved

Words," earlier in this chapter, for a complete list of the reserved words.) Also, a variable name may not be a reserved word with one of the type declaration characters ($, %, !, #) at the end. For example,

```
10 EXP = 5
```

is invalid, because EXP is a reserved word. However,

```
10 EXPONENT = 5
```

is okay, because EXP is only a part of the variable name.

A variable beginning with FN is assumed to be a call to a user-defined function (see "DEF FN Statement" in Chapter 4).

How to Declare Variable Types

A variable's name determines its type (string or numeric, and if numeric, what its precision is).

String Variables: String variable names are written with a dollar sign ($) as the last character. For example:

```
A$ = "SALES REPORT"
```

The dollar sign is a variable type declaration character. It "declares" that the variable will represent a string.

Numeric Variables: Numeric variable names may declare integer, single-, or double-precision values. Although you may get less accuracy doing computations with integer and single-precision variables, there are reasons you might want to declare a variable as a particular precision.

- Variables of higher precisions take up more room in storage. This is important if space is a consideration.

- It takes more time for the computer to do arithmetic with the higher precision numbers. A program with repeated calculations will run faster with integer variables.

The type declaration characters for numeric variables and the number of bytes required to store each type of value are as follows:

%	Integer variable (2 bytes)
!	Single-precision variable (4 bytes)
#	Double-precision variable (8 bytes)

Note: If the variable type is not explicitly declared, it defaults to single-precision.

Examples of BASIC variable names follow.

PI#	declares a double-precision value
MINIMUM!	declares a single-precision value
LIMIT%	declares an integer value
N$	declares a string value
ABC	represents a single-precision value

Variable types may also be declared in another way. The BASIC statements DEFINT, DEFSNG, DEFDBL and DEFSTR may be included in a program to declare the types for certain variable names. These statements are described in detail under "DEFtype Statements" in Chapter 4. All the examples which follow in this book assume that none of these types of declarations have been made, unless the statements are explicitly shown in the example.

Arrays

An array is a group or table of values that are referred to with one name. Each individual value in the array is

called an *element*. Array elements are variables and can be used in expressions and in any BASIC statement or function which uses variables.

Declaring the name and type of an array and setting the number of elements and their arrangement in the array is known as *defining*, or *dimensioning*, the array. Usually this is done using the DIM statement. For example:

```
10 DIM B$(5)
```

This creates a one dimensional array named B$. All of its elements are variable length strings, and the elements have an initial null value.

Array B$ could be thought of as a list of character strings, like this:

B$(0)
B$(1)
B$(2)
B$(3)
B$(4)
B$(5)

The first string in the list is named B$(0).

```
20 DIM A(2,3)
```

This creates a two-dimensional array named A. Since the name does not have a type declaration character, the array consists of single-precision values. All the array elements are initially set to 0.

Array A could be thought of as a table of rows and columns, like this:

A(0,0)	A(0,1)	A(0,2)	A(0,3)
A(1,0)	A(1,1)	A(1,2)	A(1,3)
A(2,0)	A(2,1)	A(2,2)	A(2,3)

The element in the second row, first column, is called A(1,0).

Each array element is named with the array name *subscripted* with a number or numbers. An array variable name has as many subscripts as there are dimensions in the array.

The subscript indicates the position of the element in the array. Zero is the lowest position unless you explicitly change it (see "OPTION BASE Statement" in Chapter 4). The maximum number of dimensions for an array is 255. The maximum number of elements per dimension is 32767.

If you use an array element before you define the array, it is assumed to be dimensioned with a maximum subscript of 10.

For example, if BASIC encounters the statement:

```
50 SIS(3)=500
```

and the array SIS has not already been defined, the array is set to a one-dimensional array with 11 elements, numbered SIS(0) through SIS(10). You may only use this method of *implicit declaration* for one-dimensional arrays.

One final example:

```
Ok
10 DIM YEARS(3,4)
20 YEARS(2,3)=1982
30 FOR ROW=0 TO 3
40 FOR COLUMN=0 TO 4
50 PRINT YEARS(ROW,COLUMN);
60 NEXT COLUMN
70 PRINT
80 NEXT ROW
RUN
 0  0  0  0  0
 0  0  0  0  0
 0  0  0  1982  0
 0  0  0  0  0
Ok
```

Note: A regular variable may have the same name as an array variable because A$ is different than any element in array A$ (n,...).

How BASIC Converts Numbers from One Precision to Another

When necessary, BASIC converts a number from one precision to another. The following rules and examples should be kept in mind.

- If a numeric value of one precision is assigned to a numeric variable of a different precision, the number is stored as the precision declared in the target variable name.

  Example:

```
Ok
10 A% = 23.42
20 PRINT A%
RUN
 23
Ok
```

- Rounding, as opposed to truncation, occurs when assigning any higher precision value to a lower precision variable (for example, changing from double- to single-precision values).

  Example:

```
Ok
10 C = 55.8834567#
20 PRINT C
RUN
 55.88346
Ok
```

  This affects not only assignment statements (e.g., I%=2.5 results in I%=3), but also affects function and statement evaluations (e.g., TAB(4.5) goes to the fifth position, A(1.5) is the same as A(2), and X=11.5 MOD 4 will result in a value of 0 for X).

- If you convert from a lower precision to a higher precision number, the resulting higher precision

number cannot be any more accurate than the lower precision number. For example, if you assign a single-precision value (A) to a double-precision variable (B#), only the first six digits of B# will be accurate because only six digits of accuracy were supplied with A. The error can be bounded using the following formula:

```
ABS(B#-A) < 6.3E-8 * A
```

That is, the absolute value of the difference between the printed double-precision number and the original single-precision value is less than 6.3E-8 times the original single-precision value.

Example:

```
Ok
10 A = 2.04
20 B# = A
30 PRINT A;B#
RUN
 2.04  2.039999961853027
Ok
```


- When an expression is evaluated, all the operands in an arithmetic or relational operation are converted to the same degree of precision, namely the most precise operand. Also, the result of an arithmetic operation is returned to this degree of precision.

  Examples:

```
Ok
10 D# = 6#/7
20 PRINT D#
RUN
 .8571428571428571
Ok
```

  The arithmetic is performed in double precision and the result is returned in D# as a double-precision value.

```
Ok
10 D = 6#/7
20 PRINT D
RUN
 .8571429
Ok
```

The arithmetic is performed in double precision and the result is returned to D (single-precision variable), rounded, and printed as a single-precision value.

- Logical operators (see "Logical Operators" in this chapter) convert their operands to integers and return an integer result. Operands must be in the range -32768 to 32767 or an **Overflow** error occurs.

Techniques for Formatting your Output

BASIC has built-in statements and functions that you can use in your programs to display numbers in the desired format and with the desired accuracy.

- Use DEFDBL to define your constants and variables as double-precision numbers. For example:

```
Ok
10 WIDTH 80
20 DEFDBL A
30 A=70#
40 PRINT A/100#,
50 A=A+1
60 IF A<100# GOTO 40
RUN

 .7         .71        .72        .73        .74
 .75        .76        .77        .78        .79
 .8         .81        .82        .83        .84
 .85        .86        .87        .88        .89
 .9         .91        .92        .93        .94
 .95        .96        .97        .98        .99
Ok
```

- When you want your program results displayed in decimal notation, use the PRINT USING and LPRINT USING statements. These statements let you choose the format in which the results will be printed or displayed. For example, to print up to three digits to the left of the decimal point and only one to the right, you might try the following:

```
Ok
10 WIDTH 80
20 N=100.4
30 PRINT USING "###.#    ";N;
40 N=N-2.5
50 IF N>5 GOTO 30
RUN

100.4   97.9   95.4   92.9   90.4
 87.9   85.4   82.9   80.4   77.9
 75.4   72.9   70.4   67.9   65.4
 62.9   60.4   57.9   55.4   52.9
 50.4   47.9   45.4   42.9   40.4
 37.9   35.4   32.9   30.4   27.9
 25.4   22.9   20.4   17.9   15.4
 12.9   10.4    7.9    5.4
Ok
```

Notes:

1. Avoid using both single- and double-precision numbers in the same formula because it reduces accuracy.

2. Use double-precision transcendentals for greater accuracy.

Numeric Expressions and Operators

A numeric expression may be simply a numeric constant or variable. It may also be used to combine constants and variables using operators to produce a single numeric value.

Numeric operators perform mathematical or logical operations mostly on numeric values, and sometimes on string values. We refer to them as "numeric" operators because they produce a value that is a number. The BASIC numeric operators may be divided into the following categories:

- Arithmetic
- Relational
- Logical
- Functions

Arithmetic Operators

The arithmetic operators perform the usual operations of arithmetic, such as addition and subtraction. In order of precedence, they are:

Operator	Operation	Sample Expression
^	Exponentiation	X^Y
-	Negation	-X
*, /	Multiplication, Floating Point Division	X*Y, X/Y
\	Integer Division	X\Y
MOD	Modulo Arithmetic	X MOD Y
+, -	Addition, Subtraction	X+Y, X-Y

(If you have a mathematical background, you will notice that this is the standard order of precedence.) Although most of these operations probably look familiar to you, two of them may seem a bit unfamiliar—integer division and modulo arithmetic.

Integer Division

Integer division is denoted by the backslash (\). The operands are rounded to integers (in the range -32768 to 32767) before the division is performed; the quotient is truncated to an integer.
For example:

```
Ok
10 A = 10\4
20 B = 25.68\6.99
30 PRINT A;B
RUN
 2  3
Ok
```

Modulo Arithmetic

Modulo arithmetic is denoted by the operator MOD. It gives the integer value that is the remainder of an integer division.

For example:

```
Ok
10 A = 7 MOD 4
20 PRINT A
RUN
 3
Ok
```

This result occurs because 7/4 is 1, with remainder 3.

```
Ok
PRINT 25.68 MOD 6.99
 5
Ok
```

The result is 5 because 26/7 is 3, with the remainder 5. (Remember, BASIC rounds when converting to integers.)

Relational Operators

Relational operators compare two values. The values may both be either numeric, or string. The result of the comparison is either "true" (-1) or "false" (0). This result is usually then used to make a decision regarding program flow. (See "IF Statement" in Chapter 4.)

GENERAL INFO

Operator	Relation Tested	Sample Expressions
=	Equality	X=Y
<> or ><	Inequality	X<>Y; X><Y
<	Less than	X<Y
>	Greater than	X>Y
<= or =<	Less than or equal to	X<=Y, X=<Y
>= or =>	Greater than or equal to	X>=Y, X=>Y

(The equal sign is also used to assign a value to a variable. See "LET Statement" in Chapter 4.)

Numeric Comparisons

When arithmetic and relational operators are combined in one expression, the arithmetic is always performed first. For example, the expression:

```
X+Y < (T-1)/Z
```

will be true (-1) if the value of X plus Y is less than the value of T-1 divided by Z.

More examples:

```
Ok
10 X=100
20 IF X <> 200 THEN PRINT "NOT EQUAL"
      ELSE PRINT "EQUAL"
RUN
NOT EQUAL
Ok
```

Here, the relation is true (100 is not equal to 200). The true result causes the THEN part of the IF statement to be executed.

```
Ok
PRINT 5<2; 5<10
 0 -1
Ok
```

Here the first result is false (zero) because 5 is not less than 2. The second result is -1 because the expression 5<10 is true.

String Comparisons

String comparisons can be thought of as "alphabetical." That is, one string is "less than" another if the first string comes before the other one alphabetically. Lowercase letters are "greater than" their uppercase counterparts. Numbers are "less than" letters.

The way two strings are actually compared is by taking one character at a time from each string and comparing the ASCII codes. (See Appendix G, "ASCII Chaacter Codes" for a complete list of ASCII codes.) If all the ASCII codes are the same, the strings are equal. Otherwise, as soon as the ASCII codes differ, the string with the lower code number is less than the string with the higher code number. If, during string comparison, the end of one string is reached, the shorter string is said to be smaller. Leading and trailing blanks are significant. For example, all the following relational expressions are true (that is, the result of the relational operation is -1):

```
"AA" < "AB"
"FILENAME" = "FILENAME"
"X&" > "X#"
"WR " > "WR"
"kg" > "KG"
"SMYTH" < "SMYTHE"
B$ < "718"              (where B$ = "12543")
```

All string constants used in comparison expressions must be enclosed in quotation marks.

GENERAL INFO

Logical Operators

Logical operators perform logical, or *Boolean*, operations on numeric values. Just as the relational operators are usually used to make decisions regarding program flow, logical operators are usually used to connect two or more relations and return a true or false value to be used in a decision (see "IF Statement" in Chapter 4).

A logical operator takes a combination of true-false values and returns a true or false result. An operand of a logical operator is considered to be "true" if it is not equal to zero (like the -1 returned by a relational operator), or "false" if it is equal to zero. The result of the logical operation is a number which is, again, "true" if it is not equal to zero, or "false" if it is equal to zero. The number is calculated by performing the operation bit by bit. This is explained in detail shortly.

The logical operators are NOT (logical complement), AND (conjunction), OR (disjunction), XOR (exclusive or), IMP (implication), and EQV (equivalence). Each operator returns results as indicated in the following table. ("T" indicates a true, or non-zero value. "F" indicates a false, or zero value.) The operators are listed in order of precedence.

NOT

X	***NOT X***
T	F
F	T

AND

X	***Y***	***X AND Y***
T	T	T
T	F	F
F	T	F
F	F	F

OR

X	***Y***	***X OR Y***
T	T	T
T	F	T
F	T	T
F	F	F

XOR

X	***Y***	***X XOR Y***
T	T	F
T	F	T
F	T	T
F	F	F

EQV

X	***Y***	***X EQV Y***
T	T	T
T	F	F
F	T	F
F	F	T

IMP

X	***Y***	***X IMP Y***
T	T	T
T	F	F
F	T	T
F	F	T

Some examples of ways to use logical operators in decisions:

```
IF HE>60 AND SHE<20 THEN 1000
```

Here, the result is true if the value of the variable HE is more than 60 and also the value of SHE is less than 20.

```
IF I>10 OR K<0 THEN 50
```

The result is true if I is greater than 10, or K is less than 0, or both.

```
50 IF NOT P THEN 100
```

Here, the program branches to line 100 if NOT P is true. Note that NOT P does not mean that "it is not the case that P is true" or "P is false"; it means "the NOT of P is true," which is something different. That is, NOT P does *not* produce the same result as NOT (P<>0). Refer to the next section, "How Logical Operators Work," for an explanation.

```
100 FLAG% = NOT FLAG%
```

This example switches a value back and forth from true to false.

How Logical Operators Work

Operands are converted to integers in the range -32768 to +32767. (If the operands are not in this range, an **Overflow** error results.) If the operand is negative, the two's complement form is used. This turns each operand into a sequence of 16 bits. The operation is performed on these sequences. That is, each bit of the result is determined by the corresponding bits in the two operands, according to the tables for the operator listed previously. A 1 bit is considered "true," and a 0 bit is "false."

Thus, you can use logical operators to test for a particular bit pattern. For instance, the AND operator may be used to "mask" all but one of the bits of a status byte at a machine I/O port.

The following examples will help show how the logical operators work.

```
A = 63 AND 16
```

Here, A is set to 16. Since 63 is binary 111111 and 16 is binary 10000, 63 AND 16 equals 010000 in binary, which is equal to 16.

```
B = -1 AND 8
```

B is set to 8. Since -1 is binary 11111111 11111111 and 8 is binary 1000, -1 AND 8 equals binary 00000000 00001000, or 8.

```
C = 4 OR 2
```

Here, C equals 6. Since 4 is binary 100 and 2 is binary 010, 4 OR 2 is binary 110, which is equal to 6.

```
X = 2
TWOSCOMP = (NOT X) + 1
```

This example shows how to form the two's complement of a number. X is 2, which is 10 binary. NOT X is then binary 11111111 11111101, which is -3 in decimal; -3 plus 1 is -2, the complement of 2. That is, the two's complement of any integer is the bit complement plus one.

Note that if both operands are equal to either 0 or -1, a logical operator will return either 0 or -1.

Numeric Functions

A function is used like a variable in an expression to call a predetermined operation that is to be performed on one or more operands. BASIC has "built-in" functions that reside in the system, such as SQR (square root) or SIN (sine). All of BASIC's built-in functions are listed under "Functions and Variables" in the beginning of Chapter 4. Detailed descriptions are also included in the alphabetical section of Chapter 4. You can also define your own functions using the DEF FN statement. See "DEF FN Statement" in Chapter 4.

Order of Execution

The categories of numeric operations were discussed in their order of precedence, and the precedence of each operation within a category was indicated in the discussion of the category. In summary:

1. Function calls are evaluated first

2. Arithmetic operations are performed next, in this order:

   a. ^
   b. unary -
   c. *, /
   d. \
   e. MOD
   f. +, -

3. Relational operations are done next

4. Logical operations are done last, in this order:

   a. NOT
   b. AND
   c. OR
   d. XOR
   e. EQV
   f. IMP

Operations at the same level in the list are performed in left-to-right order. To change the order in which the operations are performed, use parentheses. Operations within parentheses are performed first. Inside parentheses, the usual order of operations is maintained.

Here are some sample algebraic expressions and their BASIC counterparts.

Algebraic Expression	**BASIC Expression**
$X+2Y$	X+Y*2
$X-\frac{Y}{Z}$	X-Y/Z
$\frac{XY}{Z}$	X*Y/Z
$\frac{X+Y}{Z}$	(X+Y)/Z
$(X^2)^Y$	(X∧2)∧Y
X^{Y^Z}	X∧(Y∧Z)
X(-Y)	X*(-Y)

Note: Two consecutive operators must be separated by parentheses, as shown in the X*(-Y) example.

String Expressions and Operators

A string expression may be simply a string constant or variable, or it may combine constants and variables by using operators to produce a single string value.

String operators are used to arrange character strings in different ways. The two categories of string operators are:

- Concatenation
- Functions

Note that although you can use the relational operators =, <>, <, >, <=, and >= to compare two strings, these are not considered to be "string operators" because they produce a numeric result, not a string result. Read through "Relational Operators" earlier in this chapter for an explanation of how you can compare strings using relational operators.

Concatenation

Joining two strings together is called *concatenation*. Strings are concatenated using the plus symbol (+). For example:

```
Ok
10 COMPANY$ = "IBM"
20 TYPE$ = " PC jr"
30 FULLNAME$ = TYPE$ + " Computer"
40 PRINT COMPANY$+FULLNAME$
RUN
IBM PC jr Computer
Ok
```

String Functions

A string function is like a numeric function except that it returns a string result. A string function can be used in an expression to call a predetermined operation that is to be performed on one or more operands. BASIC has "built-in" functions that reside in the system, such as MID$, which returns a string from the middle of another string, or CHR$, which returns the character with the specified ASCII code. All of BASIC's built-in functions are listed under "Functions and Variables" in the beginning of Chapter 4. Detailed descriptions are also included in the alphabetical section of Chapter 4.

You can also define your own functions using the DEF FN statement. See "DEF FN Statement" in Chapter 4.

Input and Output

The remainder of this chapter contains information on input and output (I/O) in BASIC. The following topics are discussed:

- Files – using BASIC files, naming files, and using devices
- Paths – using BASIC paths, naming paths, and using devices
- Tree-structured directories – using tree-structured directories
- The screen – displaying things on the screen, with an emphasis on graphics
- Other features – using the clock, sound, light pen, and joysticks

Files

A file is a collection of information which is kept somewhere other than in the random access memory of the PCjr. For example, your information may be stored in a file on diskette or cassette. To use the information, you must *open* the file to tell BASIC where the information is. Then you may use the file for input and/or output.

BASIC supports the concept of general device I/O files. This means that any type of input/output may be treated like I/O to a file, whether you are actually using a cassette or diskette file, or are communicating with another computer.

File Number

BASIC performs I/O operations using a file number. The file number is a unique number that is associated with the actual physical file when it is opened. It identifies the path for the collection of data. A file number may be any number, variable, or expression ranging from 1 to *n*, where *n* is the maximum number of files allowed. The variable *n* is 4 in Cassette BASIC and defaults to 3 in Cartridge BASIC with DOS present. It may be changed up to a maximum of 15 by using the **/F:** option on the BASIC command for Cartridge BASIC (when DOS is present.)

Naming Files

The physical file is described by its *file specification*, or *filespec* for short.

The file specification is a string expression of the form:

```
device:filename
```

The device name tells BASIC *where* to look for the file, and the filename tells BASIC *which* file to look for on that particular device. Sometimes you do not need both device name and filename, so specification of device and filename is optional. Note the colon (:) indicated above. Whenever you specify a device, you *must* use the colon even though a filename is not necessarily specified.

> **Note:** File specification for communications devices is slightly different. The filename is replaced with a list of options specifying such things as line speed. Refer to "OPEN "COM... Statement" in Chapter 4 for details.

Remember that if you use a string constant for the *filespec*, you must enclose it in quotation marks. For example,

```
LOAD "A:ROTHERM.ARK"
```

A *path* consists of a list of directory names separated by backslashes (\). The path is a string expression of the form:

```
device:path
```

The device name tells BASIC *where* to look for the file, and the path tells BASIC *which* path to follow to get to the directory that contains a particular file. Sometimes you do not need both the device name and the path, so specification of *both* the device and the path is optional. Note the colon (:) indicated above. Whenever you specify a device, you *must* use the colon even though you may not specify a path.

You can use paths for the following commands:

BLOAD	KILL	OPEN
BSAVE	LOAD	RMDIR
CHAIN	MERGE	RUN
CHDIR	MKDIR	SAVE
FILES	NAME	

Notes:

1. A path may not contain more than 63 characters.

2. If you place a device name anywhere other than before the path, you will see a **Bad filename** error message.

3. If you use a string constant for the path, you must enclose it in quotation marks.

"A:\SALES\JOHN\REPORT" – is valid

\SALES\JOHN\A:.REPORT – will give an error

If a filename is included, it must also be separated from the last directory name by a backslash. If a path begins

with a backslash, BASIC starts its search from the root directory; otherwise, the search begins at the current directory.

If the file is not in the current directory, you must supply BASIC with the path of directory names leading to the current directory. The path you specify can be either the path of names starting with the root directory, or the path from the current directory.

Tree-Structured Directories

Previous releases of BASIC used a simple directory structure that was adequate for managing files on diskettes.

Cartridge BASIC gives you the ability to better organize and manage your diskettes by placing groups of related files in their own directories.

For example, let's assume that XYZ company has two departments, sales and accounting. All of the company's files are kept on the computer's diskette. The organization of the file categories could be viewed like this:

```
                 ROOT
                /    \
               /      \
           SALES     ACCOUNTING
            / |        |     \
           /  |        |      \
      MIKE    |    SHANNON    CHELLE
        |     |                  |
        |    PAM                 |
     reports                  reports
```

With Cartridge BASIC, you can create a directory structure that matches the organization above. For more information, refer to the commands MKDIR, RMDIR and CHDIR in Chapter 4.

Directory Types

As in previous releases of BASIC, a single directory is created on each diskette when you format it. That directory is called the *root* directory. A root directory on a diskette can hold either 64 or 112 files.

In addition to containing the names of files, the root directory also contains the names of other directories called *sub-directories*. Unlike the root directory, these sub-directories are actually files and can contain any number of additional files and sub-directories—limited only by the amount of available space on the diskette.

The sub-directory names are in the same format as filenames. All characters that are valid for filenames are valid for a directory name. Each directory can also contain file and directory names that also appear in other directories.

For example, using our tree-structure above, the directory called ACCOUNTING could possibly have a sub-directory called PAM at the same time that SALES has a sub-directory called PAM. Likewise, the directory SALES could also have a sub-directory called SHANNON.

Current Directory

Just as BASIC remembers a default drive, it can also remember a default directory for each drive on your system. This is called the *current* directory and is the directory that BASIC will search if you enter a filename without telling BASIC which directory the file is in. You can change the current directory by issuing the CHDIR command. (Refer to "CHDIR Command" in Chapter 4.)

Device Name

The device name consists of up to four characters followed by a colon (:). The following is a complete list of device names, telling what device the name applies to, what the device can be used for (input or output), and which versions of BASIC support the device.

Device Name Chart

KYBD: Keyboard. Input only, all versions of BASIC.

SCRN: Screen. Output only, all versions of BASIC.

LPT1: Parallel printer. Output, all versions of BASIC.

COMMUNICATIONS DEVICES

COM1: IBM Internal Modem option or (RS232) Serial Port when the Modem option is not installed.

COM2: RS232 Serial Port when the IBM Modem option is installed.

STORAGE DEVICES

CAS1: Cassette tape player. Input and output, all versions.

A: First diskette drive. Input and output, Catridge BASIC.

Refer to Appendix I, "Technical Information and Tips" for information on which adapters are referred to by the printer and communications device names.

Filename

The filename must conform to the following rules:

For cassette files:

- The name may not be more than eight characters long.
- The name may not contain colons, hex 00s or hex FFs (decimal 255s).

For diskette files, the name should conform to DOS conventions:

- The name may consist of two parts separated by a period (.):

  ```
  name.extension
  ```

  The *name* may be from one to eight characters long. The *extension* may be no more than three characters long.

  If *extension* is longer than three characters, the extra characters are truncated. If *name* is longer than eight characters and *extension* is not included, then BASIC inserts a period after the eighth character and uses the extra characters (up to three) for the *extension*. If *name* is longer than eight characters and an *extension* is included, then an error occurs.

- Only the following characters are allowed in *name* and *extension*:

  A through Z
  0 through 9
  ( ) { }
  @ # $ % ^ & !
  - _ ' ' / ~ ¦

Some examples of filenames for Cartridge BASIC are:

27HAL.DAD

VDL

PROGRAM1.BAS

$$@(!).123

The following examples show how BASIC truncates names and extensions when they are too long, as explained above.

A23456789JKLMN becomes: A2345678.9JK

@HOME.TRUM10 becomes: @HOME.TRU

SHERRYLYNN.BAS causes an error

Using the Screen

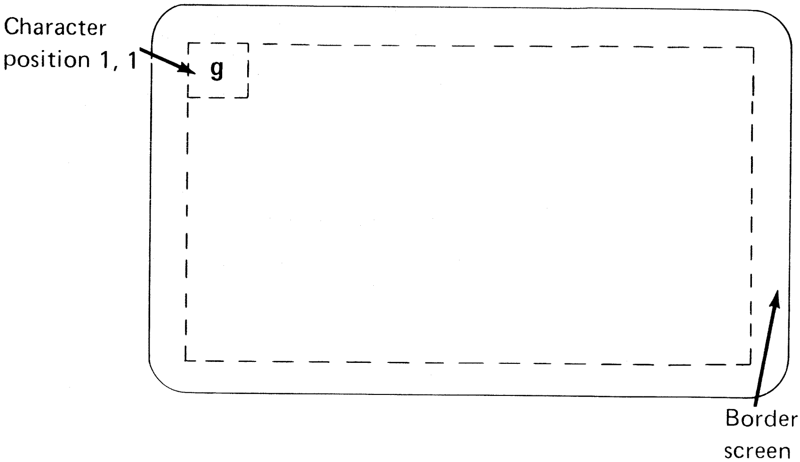
BASIC can display text, special characters, points, lines, or more complex shapes in color or in black and white.

Display

The PCjr allows you to display text in 16 different colors. (You can also display in just black and white by setting parameters on the SCREEN or COLOR or PALETTE statements.) Text refers to letters, numbers, and all the special characters in the regular character set. You have the capability to draw pictures with the special line and block characters. You can also create blinking, reverse image, invisible, highlighted, and underscored characters by setting parameters on the COLOR statement. You also get complete graphics capability to draw complex pictures. This graphics capability makes the screen *all points addressable* in low, medium and high resolution. This is more versatile than the ability to draw with the special line and block characters which you have in text mode. From now on, the term *graphics* will refer only to this special capability. The use of the extended character set with special line and block characters will not be considered graphics.

Text Mode

The screen can be pictured like this:



Characters are shown in 25 horizontal lines across the screen. These lines are numbered 1 through 25, from top to bottom. Each line has 40 character positions (or 80, depending on how you set the width). These are numbered 1 to 40 (or 80) from left to right. The position numbers are used by the LOCATE statement, and are the values returned by the POS(0) and CSRLIN functions. For example, the character in the upper left corner of the screen is on line 1, position 1. WIDTH 80, requires that your system have 128K of memory.

Characters are normally placed on the screen using the PRINT statement. The characters are displayed at the position of the cursor. Characters are displayed from left to right on each line, from line 1 to line 24. When the cursor would normally go to line 25 on the screen, lines 1 through 24 are *scrolled* up one line, so that what

was line 1 disappears from the screen. Line 24 is then blank, and the cursor remains on line 24 to continue printing.

Line 25 is usually used for "soft key" display (see "KEY Statement" in Chapter 4), but it is possible to write over this area of the screen if you turn the "soft key" display off. The 25th line is never scrolled by BASIC.

Each character on the screen is composed of two parts: foreground and background. The foreground is the character itself. The background is the "box" around the character. You can set the foreground and the background color for each character using the COLOR statement. In Cartridge BASIC you can change foreground color by using the PALETTE or PALETTE USING statement.

You can use a total of 16 different colors.

0	Black	8	Gray
1	Blue	9	Light Blue
2	Green	10	Light Green
3	Cyan	11	Light Cyan
4	Red	12	Light Red
5	Magenta	13	Light Magenta
6	Brown	14	Yellow
7	White	15	High-intensity White

The colors may vary depending on your particular display device. Adjusting the color tuning of the display may help get the colors to match this chart better.

Most television sets or monitors have an area of "overscan" which is outside the area used for characters. This overscan area is known as the *border*. You can also use the COLOR statement to set the color of the border.

The statements you can use to display information in text mode are:

CLEAR	PCOPY
CLS	PRINT
COLOR	SCREEN
LOCATE	WIDTH
PALETTE	WRITE
PALETTE USING	

The following functions and system variables may be used in text mode:

CSRLIN	SPC
POS	TAB

Another special feature you get in text mode and graphics mode is multiple display pages. The PC jr video memory is, by default, 16K. Video memory is divided into segments called pages. The size of a page is determined by the current screen mode. Text mode uses only 2K or 4K per page, depending on the width. So in a 16K memory, there would be 8 pages in 40 width, and 4 pages in 80 width. Refer to "SCREEN Statement" in Chapter 4 for details. To use WIDTH 80 you must have 128K system memory.

If you want to save some memory space because you find you do not need all 16K of the video memory, or if you find you need more space than 16K, the CLEAR statement supplies features to help you. For instance, since text mode only needs 2K or 4K depending on your width, it is possible to recover 12K or 14K of video memory with the CLEAR statement. If you increase your video memory area, you can increase the number of pages in the memory. This can be done using the CLEAR statement. The CLEAR statement is valid in all modes. For more information, refer to "CLEAR Command" in chapter 4.

Graphics Modes

You can use BASIC statements to draw in three graphic resolutions:

- low resolution: 160 by 200 points and 16 colors
- medium resolution: 320 by 200 points and 4 or 16 colors depending on the current SCREEN mode
- high resolution: 640 by 200 points and 2 or 4 colors depending on current SCREEN mode

You can select which resolution you want to use with the SCREEN statement.

The statements used for graphics in BASIC are:

CIRCLE
CLEAR
COLOR
DRAW
GET
LINE
PAINT
PALETTE
PALETTE USING
PCOPY
PRESET
PSET
PUT
SCREEN
VIEW
WINDOW

The graphics functions are:

PMAP
POINT

Attribute and Bits Per Pixel

For every point on the screen, there exists a numerical value that describes the color of each point. This numerical value is called an attribute. If you refer to the COLOR statement where all available colors are listed, you can see that the color blue is indexed by attribute 1. Refer to "Assigning Colors to Attributes" later in this section for more information.

The attribute is point information and varies depending on the current screen mode. The variance is based on the legal attribute range for each mode which is listed below. The maximum attribute for each mode determines how many bits are required to define the attribute for a point. This is called the number of bits per pixel. Consider the attribute range for screen 3 which is 0-15. In binary, four bits are required to represent a decimal number as large as 15. For a screen mode with an attribute range of 0-3, only two bits are needed to represent these values. Mathematically, the number of bits per pixel is equal to the log base 2 of the total number of colors available. This concept is particularly important when using the paint tiling feature. See the "PAINT Statement" in chapter 4 for more information.

Low Resolution: There are 160 horizontal points and 200 vertical points in low resolution. These points are numbered left to right top to bottom, starting with 0. That makes the upper left corner point (0,0), and the lower right corner point (159,199). (If you are familiar with the usual mathematical method for numbering coordinates, this may seem upside-down to you). Low resolution is set by a SCREEN 3 statement and is the only low resolution mode. You may use 16 attributes (0 to 15) in this mode. You can display text characters in any combination of colors on the screen. When you display text in low resolution, you get 20 characters on a line, 25 lines.

Medium Resolution: There are 320 horizontal points and 200 vertical points in medium resolution. These points are numbered from left to right and from top to bottom, starting with zero. That makes the upper left corner of the screen point (0,0), and the lower right corner point (319,199). Medium resolution is set by a SCREEN 1, SCREEN 4, or SCREEN 5 statement. Use of medium resolution in SCREEN 5 requires 128K of memory.

Medium resolution makes available the use of 4 or 16 attributes, depending on the current screen mode. It is possible to choose 1 of 16 colors for each attribute with the PALETTE or PALETTE USING statements. In SCREEN 1, attribute 0 is always assumed to be the background attribute. By default this is black but could be changed with the PALETTE or PALETTE USING Statements.

You may select one of two preset palettes for attributes 1, 2 and 3 in screen 1. These palettes are a set of three actual colors to be associated with the attributes 1, 2 and 3. If you change the palette with a COLOR statement, all the colors on the screen change to match the new palette. The COLOR statement is valid in all screen modes but works differently in SCREEN 1 than in the other graphics modes. You can display text in any combination of colors on the screen. When you display text in medium resolution you get 40 characters on a line, 25 lines.

High Resolution: In high resolution there are 640 horizontal points and 200 vertical points. As in medium resolution, these points are numbered starting with zero so that the lower right corner point is (639,199). High resolution is set by the SCREEN 2 or SCREEN 6 statement. Use of high resolution in SCREEN 6 requires 128K of memory.

In high resolution, SCREEN 2, there are only two colors: black and white. Black is always the 0 (zero) attribute, and white is always the 1 (one) attribute.

When you display text characters in high resolution, you get 80 characters on a line, 25 lines. The foreground attribute is 1 (one) and the background attribute is 0 (zero); so characters will always be white on black. High resolution, SCREEN 6, is different from SCREEN 2 in that it allows the use of color. You may use 4 attributes (0, 1, 2, 3) in this mode. These attributes may be reassigned any of the 16 colors through the PALETTE or PALETTE USING statements.

Assigning Colors to Attributes

Suppose you were creating a paint by number picture. You would first draw a border around the area to be colored. You would then place a number in that area to uniquely identify it. Once you had chosen the number, it would never change. This number is the attribute that uniquely identifies that area.

Selecting a color for the attribute is done with the PALETTE and PALETTE USING statements. For example, to describe a sky, we would paint an area with attribute one. We then would make the sky blue by assigning color 1 (blue) to attribute one. We could make the sky red by assigning color 4 to attribute 1. Note that any point on the screen that has attribute one will now appear red. You may use any of the 16 available colors for any one attribute by using the PALETTE statement.

Specifying Coordinates

The graphic statements require information about where on the screen you want to draw. You give this information in the form of coordinates. Coordinates are generally in the form (*x*,*y*), where *x* is the horizontal position, and *y* is the vertical position. This form is

known as *absolute form*, and refers to the actual coordinates of the point on the screen, without regard to the last point referenced.

There is another way to show coordinates, known as *relative form*. Using this form you tell BASIC where the point is relative to the last point referenced. This form looks like:

```
STEP (xoffset,yoffset)
```

You indicate inside the parentheses the *offset* in the horizontal and vertical directions from the last point referenced.

Initially, (after a change of screen in graphics mode, WIDTH, or CLS) the "last point referenced" is the point in the middle of the screen; that is (80,100) for low resolution, (160,100) for medium resolution and (320,100) for high resolution. Later graphics statements may change the last point referenced. When we discuss each statement in Chapter 4, we will indicate what each statement sets as the last point referenced.

This example shows the use of both forms of coordinates:

```
100 SCREEN 1
110 PSET (200,100) 'absolute form
120 PSET STEP (10,-20) 'relative form
```

This sets two points on the screen. Their actual coordinates are (200,100) and (210,80).

Other I/O Features

Clock

Cartridge BASIC provides the following statements and system variables:

DATE$ Ten-character string which is the system date, in the form *mm-dd-yyyy*. DOS required.

ON TIMER Traps time intervals.

TIME$ Eight-character string which indicates the time as *hh:mm:ss*. DOS required.

TIMER Indicates the number of seconds elapsed since midnight or System Reset. DOS required.

GENERAL INFO

Sound and Music

You can use the following statements to create sound on the IBM PCjr system:

BEEP Beeps the speaker.

SOUND Makes a single sound of given frequency and duration.

NOISE Generates noise through external speaker.

ON PLAY Traps play activity.

PLAY Plays music as indicated by a character string.

Light Pen

BASIC has the following statements and functions to allow input from a light pen.

PEN Function which tells whether or not the pen was triggered and gives its coordinates.

PEN Statement which enables/disables light pen function.

ON PEN Statement to trap light pen activity.

Joysticks

Joysticks can be useful in an interactive environment. BASIC supports two 2-dimensional (x and y coordinate) joysticks, or four one-dimensional paddles, each of which has a button. (Four buttons are supported only in Cartridge BASIC.) The following statements and functions are used for joysticks:

STICK Function which gives the coordinates of the joystick.

STRIG Function which gives the status of the joystick button (up or down).

STRIG Statement which enables/disables STRIG function.

ON STRIG Statement used to trap the button being pressed.

STRIG(n) Statement which enables/disables the joystick button interrupt.

Chapter 4. BASIC Commands, Statements, Functions, and Variables

Contents

STATEMENTS

STATEMENTS

How to Use This Chapter

This chapter has descriptions of all the BASIC commands, statements, functions, and variables. BASIC's built-in functions and variables may be used in any program without further definition.

The first pages contain a list of all the commands, statements, functions, and variables. These lists may be useful as a quick reference. The rest of the chapter describes each command, statement, function, and variable in more detail.

The distinction between a command and a statement is largely a matter of tradition. Commands, because they generally operate on programs, are usually entered in direct mode. Statements generally direct program flow from within a program, and so are usually entered in indirect mode as part of a program line. Actually, most BASIC commands and statements can be entered in either direct or indirect mode.

The description of each command, statement, function, or variable in this chapter is formatted as follows:

Purpose: Tells what the command, statement, function, or variable does.

Versions: Indicates which versions of BASIC allow the command, statement, function, or variable. For example, if you look under "CHAIN Statement" in this chapter, you will note that after **Versions:** it says:

Cassette	Cartridge	Compiler
	***	(**)

The asterisks indicate which versions of BASIC support the function. This example shows that you can use the CHAIN statement for programs written in the Cartridge version of BASIC.

In this example you will notice that the asterisks under the word "Compiler" are in parentheses. This means that there are differences between the way the statement works under the BASIC interpreter and the way it works under the IBM Personal Computer BASIC Compiler. The IBM Personal Computer BASIC Compiler is an optional software package available from IBM. If you have the BASIC Compiler, the IBM Personal Computer *BASIC Compiler* manual explains these differences.

Format: Shows the correct format for the command, statement, function, or variable. A complete explanation of the syntax format is presented in the Preface. Keep these rules in mind:

- Words in capital letters are keywords and must be entered as shown. They may be entered in any combination of uppercase and lowercase letters. BASIC always converts words to uppercase (unless they are part of a quoted string, remark, or DATA statement).

- You are to supply any items in lowercase italic letters.

- Items in square brackets [ ] are optional.

- An ellipsis (...) indicates that an item may be repeated as many times as you wish.

- All punctuation except square brackets (such as commas, parentheses, semicolons, hyphens, or equal signs) must be included where shown.

Remarks: Describes in detail how the command, statement, function, or variable is used.

Example: Shows direct mode statements, sample programs, or program segments that demonstrate the use of the command, statement, function, or variable.

In the formats given in this chapter, some of the parameters have been abbreviated as follows:

x, y, z	represent any numeric expressions
i,j,k,m,n	represent integer expressions
x\$, y\$	represent string expressions
v, v\$	represent numeric and string variables, respectively

If a single- or double-precision value is supplied where an integer is required, BASIC rounds the fractional portion and uses the resulting integer.

Functions and Variables: In the format description, most of the functions and variables are shown on the right side of an assignment statement. This is to remind you that they are not used like statements and commands. It is not meant to suggest that you are limited to using them in assignment statements. You can use them anywhere you would use a regular variable, *except* on the left side of an assignment statement. Any exceptions are noted in the particular section describing the function or variable. A few of the functions are limited to being used in PRINT statements; these are shown as part of a PRINT statement.

Commands

The following is a list of all the commands used in BASIC. The syntax of each command is shown, but not always in its entirety. You can find detailed information about each command in the alphabetical part of this chapter. You may also want to check the next section in this chapter, "Statements," for a list of the BASIC statements.

Command	Action
AUTO number,increment	Generates line numbers automatically.
BLOAD filespec,offset	Loads binary data (such as a machine language program) into memory.
BSAVE filespec,offset,length	Saves binary data.
CHDIR path	Changes the current directory.
CLEAR m,s,v	Clears program variables, and optionally sets memory area.
CONT	Continues program execution.
DELETE line1-line2	Deletes specified program lines.
EDIT line	Displays a program line for changing.
FILES filespec	Lists files in the diskette directory that match a file specification.
KILL filespec	Erases a diskette file.

LIST line1-line2,filespec
Lists program lines on the screen or to the specified file.

LLIST line1-line2
Lists program lines on the printer.

LOAD filespec Loads a program file. Can include the **R** option to run it.

MERGE filespec Merges a saved program with the program in memory.

MKDIR path Creates a directory on the specified diskette.

NAME filespec AS filename
Renames a diskette file.

NEW Erases the current program and variables.

RENUM newnum,oldnum,increment
Renumbers program lines.

RESET Reinitializes diskette information. Similar to CLOSE.

RMDIR path Removes a specified sub-directory from an existing directory.

RUN filespec Executes a program. The **R** option may be used to keep files open.

RUN line Runs the program in memory starting at the specified line.

SAVE filespec Saves the program in memory under the given filename. **A** or **P** option saves in ASCII or protected format.

SYSTEM	Ends BASIC. Closes all files and returns to DOS.
TRON, TROFF	Turns trace on or off.

Statements

This section lists all the BASIC statements alphabetically in two categories: I/O (Input/Output) Statements and Non-I/O Statements. The list tells what each statement does and shows the syntax. For the more complex statements the syntax shown may not be complete. You can find detailed information about each statement in the alphabetical portion of this chapter, later on.

You may also want to look at the previous section, "Commands," for a list of the BASIC commands.

Non-I/O Statements

Statement	Action
CALL numvar(variable list)	Calls a machine language program.
CHAIN filespec	Calls a program and passes variables to it. Other options allow you to use overlays, begin running at a line other than the first line, pass all variables, or delete an overlay.
COM(n) ON/OFF/STOP	Enables and disables trapping of communications activity.
COMMON list of variables	Passes variables to a chained program.
DATE$ = x$	Sets the date.

STATEMENTS

DEF FNname(arg list)=definition
Defines a numeric or string function.

DEFtype ranges of letters
Defines default variable types, where *type* is INT, SNG, DBL, or STR.

DEF SEG=address Defines current segment of memory.

DEF USRn=offset Defines starting address for machine language subroutine n.

DIM list of subscripted variables
Declares maximum subscript values for arrays and allocates space for them.

END Stops the program, closes all files, and returns to command level.

ERASE arraynames
Eliminates arrays from a program.

ERROR n Simulates error number n.

FOR variable=x TO y STEP z
Repeats program lines a number of times. The NEXT statement closes the loop.

GOSUB line Calls a subroutine by branching to the specified line. The RETURN statement returns from the subroutine.

GOTO line Branches to the specified line.

IF expression THEN clause ELSE clause
Performs the statement(s) in the THEN clause if expression is true (nonzero). Otherwise, performs the ELSE clause or goes to the next line.

KEY ON/OFF/LIST
Displays soft keys, turns display off, or lists key values.

**KEY n, x$** Sets soft key n to the value of the string x$.

KEY(n) ON/OFF/STOP
Enables/disables trapping of function keys or cursor control keys.

LET variable=expression
Assigns the value of the expression to the variable.

**MID$(v$,n,m)=y$** Replaces part of the variable v$ with the string y$, starting at position n and replacing m characters.

MOTOR state Turns cassette motor on if state is nonzero, off if state is zero.

NEXT variable Closes a FOR...NEXT loop (see FOR).

ON COM(n) GOSUB line
Enables trap routine for communications activity.

ON ERROR GOTO line
Enables error trap routine beginning at line specified.

ON n GOSUB line list
Branches to subroutine specified by n.

ON n GOTO line list
Branches to statement specified by n.

ON KEY(n) GOSUB line
Enables trap routine for the specified function key or cursor control key.

ON PEN GOSUB line
Enables trap routine for light pen.

ON PLAY(n) GOSUB line
Enables trap routine for play activity.

ON STRIG(n) GOSUB line
Enables trap routine for joystick button.

ON TIMER(n) GOSUB line
Enables trap routine for time intervals.

OPTION BASE n Specifies the minimum value for array subscripts.

PEN ON/OFF/STOP Enables/disables the light pen function.

POKE n,m Puts byte m into memory at the location specified by n.

RANDOMIZE n Reseeds the random number generator.

REM remark Includes remark in program.

RESTORE line	Resets DATA pointer so DATA statements may be reread.
RESUME line/NEXT/0	Returns from error trap routine.
RETURN line	Returns from subroutine.
STOP	Stops program execution, prints a break message, and returns to command level.
STRIG ON/OFF	Enables/disables joystick button function.
STRIG(n) ON/OFF/STOP	Enables/disables joystick button trapping.
SWAP variable1,variable2	Exchanges values of two variables.
TERM	If you are using Cartridge BASIC and have the proper communication device (IBM Internal Modem or external modem), then you can enter into a Terminal Emulation program via the TERM statement. For more information refer to the TERM statement in this Chapter.
TIME$ = x$	Sets the time.
v = TIMER	Returns the number of seconds since midnight or last System Reset.
WAIT port,n,m	Suspends program execution until the specified port develops the specified bit pattern.
WEND	Closes a WHILE...WEND loop (see WHILE).

WHILE expression

Begins a loop which executes when the expression is true.

I/O Statements

Statement	Action
BEEP	Beeps the speaker. Options enable/disable sound to the internal/external speaker when used in conjunction with SOUND ON/SOUND OFF.
CIRCLE (x,y),r	Draws a circle with center (x,y) and radius r. Other options allow you to specify a part of the circle to be drawn, or to change the aspect ratio to draw an ellipse.
CLOSE #f	Closes a file.
CLS	Clears the screen.

COLOR foreground,background,border

In text mode, sets colors for foreground, background, and the border screen.

COLOR foreground,background

In graphics modes 3-6 sets color for foreground,background.

COLOR background,palette

In graphics mode, (SCREEN 1) sets background color and palette of foreground colors.

DATA list of constants

Creates a data table to be used by READ statements.

DRAW string Draws a figure as specified by string.

FIELD #f,width AS stringvar...
Defines fields in a random file buffer.

GET #f,number Reads a record from a random file.

GET (x1,y1)-(x2,y2),arrayname
Reads graphic information from screen.

INPUT "prompt";variable list
Reads data from the keyboard.

INPUT #f,variable list
Reads data from file f.

LINE (x1,y1)-(x2,y2)
Draws a line on the screen. Other parameters allow you to draw a box, fill in the box and do line-styling.

LINE INPUT "prompt";stringvar
Reads an entire line from the keyboard, ignoring commas or other delimiters.

LINE INPUT #f,stringvar
Reads an entire line from a file.

LOCATE row,col Positions the cursor. Other parameters allow you to define the size of the cursor and whether it is visible or not.

LPRINT list of expressions
Prints data on the printer.

LPRINT USING v$;list of expressions
Prints data on the printer using the format specified by v$.

LSET stringvar=x$
Left-justifies a string in a field.

NOISE source,volume,duration
Generates noise through external speaker.

OPEN filespec FOR mode AS #f
Opens the file for the mode specified. You can also specify a path to be followed. Another option sets the record length for random files.

OPEN mode,#f,filespec,recl
Alternative form of preceding OPEN.

OPEN "COMn:options" AS #f
Opens file for communications.

OUT n,m
Outputs the byte m to the machine port n.

PAINT (x,y),paint,boundary,background
Fills in an area on the screen defined by boundary with the paint color.

PALETTE attribute,color
Allows control of hardware palette.

PALETTE USING arrayname (starting index)
Allows setting of all palette entries with one statement.

PCOPY source,destination
Allows copying of one page to another page.

PLAY string
Plays music as specified by string.

PRINT list of expressions
Displays data on the screen.

PRINT USING v$;list of expressions
Displays data using the format specified by v$.

PRINT #f, list of exps
Writes the list of expressions to file f.

PRINT #f, USING v$;list of exps
Writes data to file f using the format specified by v$.

PRESET (x,y) Draws a point on the screen in background color. See PSET.

PSET (x,y),attribute
Draws a point on the screen, in the foreground color if attribute is not specified.

PUT #f,number Writes data from a random file buffer to the file.

PUT (x,y),array,action
Writes graphic information to the screen.

READ variable list
Retrieves information from the data table created by DATA statements.

RSET stringvar=x$
Right-justifies a string in a field. See LSET.

SCREEN mode,burst,apage,vpage,erase
Sets screen mode, color on or off, display page, active page, and amount of video memory to be erased.

SOUND freq,duration,volume,voice
Generates sound through the speaker.

VIEW (x1,y1)-(x2,x2),attribute,boundary
Defines a viewport within the actual limits of the screen.

WIDTH size
Sets screen width. Other options allow you to specify the width of a printer or a communications file.

WINDOW (x1,y1)-(x2,y2)
Defines transformation between upper-left coordinates and lower-right coordinates.

WRITE list of expressions
Outputs data on the screen.

WRITE #f, list of expressions
Outputs data to a file.

Functions and Variables

The built-in functions and variables available in BASIC are listed below, grouped into two general categories: numeric functions, or those which return a numeric result; and string functions, or those which return a string result.

Each category is further subdivided according to the usage of the functions. The numeric functions are divided into general arithmetic (or algebraic) functions; string-related functions, which operate on strings; and input/output and miscellaneous functions. The string functions are separated into general string functions, and input/output and miscellaneous string functions.

Numeric Functions (return a numeric value)

ARITHMETIC

Function	Result
ABS(x)	Returns the absolute value of x.
ATN(x)	Returns the arctangent (in radians) of x.
CDBL(x)	Converts x to a double-precision number.
CINT(x)	Converts x to an integer by rounding.
COS(x)	Returns the cosine of angle x, where x is in radians.
CSNG(x)	Converts x to a single-precision number.
EXP(x)	Raises *e* to the x power.

STATEMENTS

FIX(x)	Truncates x to an integer.
INT(x)	Returns the largest integer less than or equal to x.
LOG(x)	Returns the natural logorithm of x.
RND(x)	Returns a random number.
SGN(x)	Returns the sign of x.
SIN(x)	Returns the sine of angle x, where x is in radians.
SQR(x)	Returns the square root of x.
TAN(x)	Returns the tangent of angle x, where x is in radians.

STRING-RELATED

Function	**Result**
ASC(x$)	Returns the ASCII code for the first character in x$.
CVI(x$), CVS(x$), CVD(x$)	Converts x$ to a number of the indicated precision.
INSTR(n,x$,y$)	Returns the position of first occurrence of y$ in x$ starting at position n.
LEN(x$)	Returns the length of x$.
VAL(x$)	Returns the numeric value of x$.

I/O and MISCELLANEOUS

Function	Result
CSRLIN	Returns the vertical line position of the cursor.
EOF(f)	Indicates an end of file condition on file f.
ERL	Returns the line number where the last error occurred (see ERR).
ERR	Returns the error code number of the last error.
FRE(x$)	Returns the amount of free space in memory not currently in use by BASIC.
INP(n)	Reads a byte from port n.
LOC(f)	Returns the "location" of file f: • next record number of random file • number of sectors read or written for sequential file • number of characters in communications input buffer
LOF(f)	Returns the length of file f: • number of bytes in sequential or random file • number of bytes free in communications input buffer
LPOS(n)	Returns the carriage position of the printer.

PEEK(n)	Reads the byte in memory location n.
PEN(n)	Reads the light pen.
PLAY(n)	Returns the number of notes in the music background buffer.
PMAP	Maps actual and relative coordinates.
POINT(x,y)	Returns the color of point (x,y) (graphics mode).
POINT(n)	Returns the value of the current x or y coordinate.
POS(n)	Returns the cursor column position.
SCREEN(row,col,z)	Returns the character or color at position (row,col).
STICK(n)	Returns the coordinates of a joy stick.
STRIG(n)	Returns the state of a joy stick button.
USRn(x)	Calls a machine language subroutine with argument x.
TIMER	Returns the number of seconds since midnight or System Reset.
VARPTR(variable)	Returns the address of the variable in memory.
VARPTR(#f)	Returns the address of the file control block for file f.

String Functions (return a string value)

GENERAL

Function	Result
CHR$(n)	Returns the character with ASCII code n.
LEFT$(x$,n)	Returns the leftmost n characters of x$.
MID$(x$,n,m)	Returns m characters from x$ starting at position n.
RIGHT$(x$,n)	Returns the rightmost n characters of x$.
SPACE$(n)	Returns a string of n spaces.
STRING$(n,m)	Returns the character with ASCII value m, repeated n times.
STRING$(n,x$)	Returns the first character of x$ repeated n times.

I/O and MISCELLANEOUS

Function	Result
DATE$	Returns the system date.
HEX$(n)	Converts n to a hexadecimal string.
INKEY$	Reads a character from the keyboard.
INPUT$(n,#f)	Reads n characters from file f.
MKI$(x), MKS$(x), MKD$(x)	Converts x in indicated precision to proper length string.

OCT$(n)	Converts n to an octal string.
SPC(n)	Prints n spaces in a PRINT or LPRINT statement.
STR$(x)	Converts x to a string value.
TAB(n)	Tabs to position n in a PRINT or LPRINT statement.
TIME$	Returns the system time.
VARPTR$(v)	Returns a three-byte string containing the type of variable, and the address of the variable in memory.

Purpose: Returns the absolute value of the expression *x*.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: *v* = ABS(*x*)

Remarks: *x* may be any numeric expression.

The absolute value of a number is always positive or zero.

Example:

```
Ok
PRINT ABS(7*(-5))
 35
Ok
```

The absolute value of -35 is positive 35.

ASC Function

Purpose: Returns the ASCII code for the first character of the string *x$*.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: *v* = ASC(*x$*)

Remarks: *x$* may be any string expression.

The result of the ASC function is a numerical value that is the ASCII code of the first character of the string *x$*. (See Appendix G, "ASCII Character Codes," for ASCII codes.) If *x$* is null, an **Illegal function call** error is returned.

The CHR$ function is the inverse of the ASC function, and it converts the ASCII code to a character.

Example:

```
Ok
10 X$ = "TEST"
20 PRINT ASC(X$)
RUN
 84
Ok
```

This example shows that the ASCII code for a capital T is 84. **PRINT ASC("TEST")** would work just as well.

Purpose: Returns the arctangent of *x*.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: *v* = ATN(*x*)

Remarks:

x may be a numeric expression of any type.

The ATN function returns the angle whose tangent is *x*. The result is a value in radians in the range -PI/2 to PI/2, where PI=3.141593. If you want to convert radians to degrees, multiply by 180/PI.

The value of ATN is calculated in single precision in Cassette BASIC and in either single or double precision in Cartridge BASIC.

Example: The first example below calculates the arctangent of 3. The second example finds the angle whose tangent is 1. It is .7853983 radians, or 45 degrees.

```
Ok
PRINT ATN(3)
 1.249046
Ok

10 RADIANS=ATN(1)
20 PI=3.141593: DEGREES=RADIANS*180/PI
30 PRINT RADIANS,DEGREES
RUN
 .7853983      45
Ok
```

AUTO
Command

Purpose: Generates a line number automatically each time you press Enter.

Versions:

Cassette	Cartridge	Compiler
***	***	

Format: AUTO [*number*] [,[*increment*]]

Remarks:

number is the number which will be used to start numbering lines. A period (.) may be used in place of the line number to indicate the current line.

increment is the value that will be added to each line number to get the next line number.

Numbering begins at *number* and increments each subsequent line number by *increment*. If both values are omitted, the default is 10,10. If *number* is followed by a comma but *increment* is not specified, the last increment specified in an AUTO command is assumed. If *number* is omitted but *increment* is included, then line numbering begins with 0.

AUTO is usually used for entering programs. It spares you from having to type each line number.

If AUTO generates a line number that already exists in the program, an asterisk (*) is printed after the number to warn you that any input will replace the existing line. However, if you press Enter immediately after the asterisk, the existing line will not be replaced and AUTO will generate the next line number.

AUTO ends when you press the Fn key followed by the Break key. The line in which the Fn and Break keys are typed is not saved. After you press the Fn and Break keys, BASIC returns to command level.

> **Note:** When in AUTO mode, you may make changes only to the current line. If you want to change another line on the screen, be sure to exit AUTO by first pressing the Fn key followed by the Break key.

Example:

```
AUTO
```

This command generates line numbers 10, 20, 30, 40, ...

```
AUTO 100,50
```

This generates line numbers 100, 150, 200, ...

```
AUTO 500,
```

This generates line numbers 500, 550, 600, 650, ... The increment is 50 since 50 was the increment in the previous AUTO command.

```
AUTO ,20
```

This generates line numbers 0, 20, 40, 60, ...

BEEP
Statement

Purpose: Causes the speaker to beep.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: BEEP

BEEP ON

BEEP OFF

Remarks: The BEEP statement sounds the speaker at 800 Hertz (cycles per second) for 1/4 second. BEEP has the same effect as:

```
PRINT CHR$(7);
```

In Cartridge BASIC the BEEP statement can be used with the SOUND statement to specify where to direct sound. It may be directed to the internal speaker and/or the external speaker.

```
BEEP ON : SOUND OFF
```

This sends the sound source through the television/external speaker and the internal speaker.

```
BEEP OFF : SOUND OFF
```

This sends the sound only through the internal speaker.

BASIC will restore the machine to the default BEEP ON/SOUND OFF state when a RUN command is executed.

BEEP Statement

Refer to "SOUND Statement" in this chapter for an explanation of SOUND with the external speaker.

Example:

```
2430 IF X < 20 THEN BEEP
```

In this example, the program checks to see if X is less than 20 (out of range). If it is, the computer "complains" by beeping.

BLOAD Command

Purpose: Loads a memory image file into memory.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: BLOAD *filespec* [,*offset*]

Remarks:

filespec is a string expression for the file specification. It must conform to the rules outlined under "Naming Files" in Chapter 3, otherwise a **Bad file name** error occurs and the load is canceled.

offset is a numeric expression in the range 0 to 65535. This is the address at which loading is to start, specified as an offset into the segment declared by the last DEF SEG statement.

If *offset* is omitted, the offset specified at BSAVE is assumed. That is, the file is loaded into the same location it was saved from.

When a BLOAD command is executed, the named file is loaded into memory starting at the specified location. If the file is to be loaded from the device CAS1:, the cassette motor is turned on automatically.

If you are using Cassette BASIC and the device name is omitted, CAS1: is assumed. CAS1: is the only allowable device for BLOAD in Cassette BASIC and in Cartridge BASIC when DOS is not

used. If you are using Cartridge BASIC with DOS and the device name is omitted, the DOS default diskette drive is assumed.

BLOAD is intended to be used with a file that has previously been saved with BSAVE. BLOAD and BSAVE are useful for loading and saving machine language programs. (You may perform machine language programs from within a BASIC program by using the CALL statement.) However, BLOAD and BSAVE are not restricted to machine language programs. Any segment may be specified as the target or source for these statements via the DEF SEG statement. You have a useful way of saving and displaying screen images: save from or load to the screen buffer.

> **Warning:** BASIC does not do any checking on the address. That is, it is possible to BLOAD anywhere in memory. You should not BLOAD over BASIC's stack, BASIC's variable area, or your BASIC program.

Notes when using CAS1:

1. If you enter the BLOAD command in direct mode, the file names on the tape will be displayed on the screen followed by a period (.) and a single letter indicating the type of file. This is followed by the message **Skipped** for the files not matching the named file, and **Found** when the named file is found. Types of files and the associated letter are:

   .B for BASIC programs in internal format (created with SAVE command)

.P for protected BASIC programs in internal format (created with SAVE ,P command)

.A for BASIC programs in ASCII format (created with SAVE ,A command)

.M for memory image files (created with BSAVE command)

.D for data files (created by OPEN followed by output statements)

If the BLOAD command is executed in a BASIC program, the file names skipped and found are not displayed on the screen.

2. You may press the Fn key followed by the Break key any time during BLOAD. This will cause BASIC to stop the search and return to direct mode between files or after a time-out period. Previous memory contents do not change.

3. If CAS1: is specified as the device and the filename is omitted, the next memory image (.M) file on the tape is loaded.

Example:

```
10 'load the screen buffer
20 'point SEG at screen buffer
30 DEF SEG= &HB800
40 'load PICTURE into screen buffer
50 BLOAD "PICTURE",0
```

This example loads the screen buffer which is at absolute address hex B8000. Note that the DEF SEG statement in 30 and the offset of 0 in 50 is wise. This ensures that the correct address is used.

The example for BSAVE in the next section illustrates how PICTURE was saved.

BSAVE Command

Purpose: Saves portions of the computer's memory on the specified device.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: BSAVE *filespec,offset,length*

Remarks:

filespec is a string expression for the file specification. It must conform to the rules outlined under "Naming Files" in Chapter 3; otherwise, a **Bad file name** error occurs and the save is canceled.

offset is a numeric expression in the range 0 to 65535. This is the offset into the segment declared by the last DEF SEG. Saving will start from this position.

length is a numeric expression in the range 1 to 65535. This is the length of the memory image to be saved.

If *offset* or *length* is omitted, a **Syntax error** occurs and the save is canceled.

If the device name is omitted in Cassette BASIC, CAS1: is assumed. CAS1: is the only allowable device for BSAVE in Cassette BASIC and in Cartridge BASIC when DOS is not used. If you are using Cartridge BASIC with DOS and the device name is omitted, the DOS default diskette drive is assumed.

If you are saving to CAS1:, the cassette motor will be turned on and the memory image file will be immediately written to the tape.

BLOAD and BSAVE are useful for loading and saving machine language programs (which can be called using the CALL statement). However, BLOAD and BSAVE are not restricted to machine language programs. By using the DEF SEG statement, any segment may be specified as the target or source for these statements. For example, you can save an image of the screen by doing a BSAVE of the screen buffer.

Example:

```
10 'Save the color screen buffer
15 'point segment at screen buffer
20 DEF SEG= &HB800
25 'save buffer in file PICTURE
30 BSAVE "PICTURE",0,&H4000
```

As explained in the example for BLOAD in the previous section, the address of the 16K screen buffer is hex B8000.

The DEF SEG statement must be used to set up the segment address to the start of the screen buffer. Offset of 0 and length &H4000 specifies that the entire 16K screen buffer is to be saved.

CALL
Statement

Purpose: Calls a machine language subroutine.

Versions:

Cassette	Cartridge	Compiler
***	***	(**)

Format: CALL *numvar* [(*variable* [,*variable*]...)]

Remarks:

numvar is the name of a numeric variable. The value of the variable indicates the starting memory address of the subroutine being called as an offset into the current segment of memory (as defined by the last DEF SEG statement).

variable is the name of a variable which is to be passed as an argument to the machine language subroutine.

The CALL statement is one way of interfacing machine language programs with BASIC. The other way is by using the USR function. Refer to Appendix C, "Machine Language Subroutines" for specific considerations about using machine language subroutines.

Example:

```
100 DEF SEG=&H1800
110 OZ=0
120 CALL OZ(A,B$,C)
```

Line 100 sets the segment to location hex 18000. OZ is set to zero so that the call to OZ will execute the subroutine at location hex 18000. The variables A, B$, and C are passed as arguments to the machine language subroutine.

CDBL Function

Purpose: Converts *x* to a double-precision number.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: *v* = CDBL(*x*)

Remarks: *x* may be any numeric expression.

Rules for converting from one numeric precision to another are followed as explained in "How BASIC Converts Numbers from One Precision to Another" in Chapter 3. Refer also to the CINT and CSNG functions for converting numbers to integer and single precision.

Example:

```
Ok
10 A = 454.67
20 PRINT A;CDBL(A)
 Ok
RUN
 454.67  454.6700134277344
```

The value of CDBL(A) is only accurate to the second decimal place after rounding. This is because only two decimal places of accuracy were supplied with A.

Purpose: Transfers control to another program, and passes variables to it from the current program.

> **Note:** This statement requires the use of DOS 2.10. If DOS 2.10 is not present, then an **Illegal function call** error will occur.

Versions:

Cassette	Cartridge	Compiler
	***	(**)

Format: CHAIN [MERGE] *filespec* [,[*line*] [,[ALL] [,DELETE *range*]]]

Remarks: *filespec* follows the rules for file specifications outlined in "Naming Files" in Chapter 3. The filename is the name of the program that is transferred to. Example:

```
CHAIN "A:PROG1"
```

line is a line number or an expression that evaluates to a line number in the chained-to program. It specifies the line at which the chained-to program is to begin running. If it is omitted, execution begins at the first line in the chained-to program. Example:

```
CHAIN "A:PROG1",1000
```

line (1000 in this example) is not affected by a RENUM command. If PROG1 is renumbered, this example CHAIN statement should be changed to point to the new line number.

ALL specifies that every variable in the current program is to be passed to the chained-to program. If the ALL option is omitted, you must include a COMMON statement in the chaining program to pass variables to the chained-to program. See "COMMON Statement" in this chapter. Example:

```
CHAIN "A:PROG1",1000,ALL
```

MERGE brings a section of code into the BASIC program as an overlay. That is, a MERGE operation is performed with the chaining program and the chained-to program. The chained-to program must be an ASCII file if it is to be merged. Example:

```
CHAIN MERGE "A:OVRLAY",1000
```

After using an overlay, you will usually want to delete it so that a new overlay may be brought in. To do this, use the DELETE option, which behaves like the DELETE command. As in the DELETE command, the line numbers specified as the first and last line of the range must exist, or an **Illegal function call** error occurs. Example:

```
CHAIN MERGE "A:OVRLAY2",1000,DELETE 1000-5000
```

This example will delete lines 1000 through 5000 of the chaining program before loading in the overlay (chained-to program). The line numbers in *range* are affected by the RENUM command.

Notes:

1. The CHAIN statement leaves files open.

2. The CHAIN statement with MERGE option preserves the current OPTION BASE setting.

3. Without MERGE, CHAIN does not preserve variable types or user-defined functions for use by the chained-to program. That is, any DEFINT, DEFSNG, DEFDBL, DEFSTR, or DEF FN statements containing shared variables must be restated in the chained program.

4. The CHAIN statement does a RESTORE before running the chained-to program. This means that the read operation does not continue where it left off in the chaining program. The next READ statement accesses the first item in the first DATA statement encountered in the program.

CHDIR Command

Purpose: Allows you to change the current directory.

Note: This command requires the use of DOS 2.10.

Versions: Cassette Cartridge *** Compiler

Format: CHDIR *path*

Remarks:

path is a string expression, not exceeding 63 characters, identifying the new directory that will become the current directory. For more information on paths refer to "Naming Files" and "Tree-Structured Directories" in Chapter 3.

Examples:

```
                ROOT
                /  \
               /    \
          SALES    ACCOUNTING
           /          /     \
          /          /       \
      MIKE      SHANNON     CHELLE
       /
      /
    PAM
```

The following examples refer to the tree-structure above.

To change to the root directory from any sub-directory, use:

```
CHDIR "\"
```

To change to the directory PAM from the root directory, use:

```
CHDIR "SALES\MIKE\PAM"
```

To change to the directory CHELLE from the directory ACCOUNTING, use:

```
CHDIR "CHELLE"
```

To change from the directory MIKE to the directory SALES, use:

```
CHDIR ".."
```

CHR$ Function

Purpose: Converts an ASCII code to its character equivalent.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: *v$* = CHR$(*n*)

Remarks: *n* must be in the range 0 to 255.

The CHR$ function returns the one-character string with ASCII code *n*. (ASCII codes are listed in Appendix G, "ASCII Character Codes.") CHR$ is commonly used to send a special character to the screen or printer. For instance, the BEL character, which beeps the speaker, might be included as CHR$(7) as a preface to an error message (instead of using BEEP). Look under "ASC Function," earlier in this chapter, to see how to convert a character back to its ASCII code.

Example:

```
Ok
PRINT CHR$(66)
B
Ok
```

The next example sets function key Fn+1 to the string "AUTO" joined with Enter. This is a good way to set the function keys so the Enter is automatically done for you when you press the function key.

```
Ok
KEY 1,"AUTO"+CHR$(13)
Ok
```

The following example is a program which shows all the displayable characters, along with their ASCII codes, on the screen.

```
10 CLS
20 FOR I=1 TO 255
30 ' ignore nondisplayable characters
40 IF (I>6 AND I<14) OR (I>27 AND I<32) THEN 100
50 COLOR 0,7 ' black on white
60 PRINT USING "###"; I ; ' 3-digit ASCII code
70 COLOR 7,0 ' white on black
80 PRINT " "; CHR$(I); " ";
90 IF POS(0)>75 THEN PRINT ' go to next line
100 NEXT I
```

CINT Function

Purpose: Converts *x* to an integer.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: *v* = CINT(*x*)

Remarks:

x may be any numeric expression. If *x* is not in the range -32768 to 32767, an **Overflow** error occurs.

x is converted to an integer by rounding the fractional portion.

See the FIX and INT functions, both of which also return integers. See also the CDBL and CSNG functions for converting numbers to single or double precision.

Example:

```
Ok
PRINT CINT(45.67)
 46
Ok
PRINT CINT(-2.89)
-3
Ok
```

Observe in both examples how rounding occurs.

Purpose: To draw an ellipse on the screen with center (*x,y*) and radius *r*.

Versions:

Cassette	Cartridge	Compiler
	***	***

Graphics mode only.

Format: CIRCLE (*x,y*),*r* [,*attribute* [,*start,end* [,*aspect*]]]

Remarks:

(x,y) are the coordinates of the center of the ellipse. The coordinates may be given in either absolute or relative form. See "Specifying Coordinates" under "Graphics Modes" in Chapter 3.

r is the radius (major axis) of the ellipse in points.

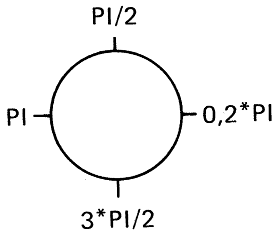
attribute is an integer or an integer expression that specifies the attribute of the ellipse, in the range of 0 to 15. In low resolution there are 16 attributes available (0 to 15). In medium resolution, there are 4 (0 to 3) or 16 (0 to 15) attributes available, depending on the current screen mode. In high resolution there are 2 (0 and 1) or 4 (0 to 3) attributes available, depending on the current screen mode. The default attribute is always the maximum attribute for the current screen mode. 0 is the background attribute. (For more information see "Graphics Mode" in Chapter 3).

Note: Screen modes 3-6 are not supported in the BASIC Compiler.

start, end are angles in radians and may range from -2*PI to 2*PI, where PI=3.141593.

aspect is a numeric expression.

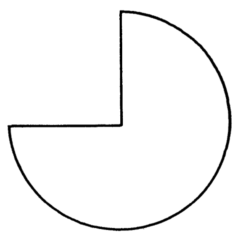
start and *end* specify where the drawing of the ellipse will begin and end. The angles are positioned in the standard mathematical way, with 0 to the right and going counterclockwise:



If the start or end angle is negative (-0 is not allowed), the ellipse will be connected to the center point with a line, and the angles will be treated as if they were positive (note that this is not the same as adding 2*PI). The start angle may be greater or less than the end angle.
For example,

```
10 PI=3.141593
20 SCREEN 1
30 CIRCLE (160,100),60,,-PI,-PI/2
```

draws a part of a circle similar to the following:

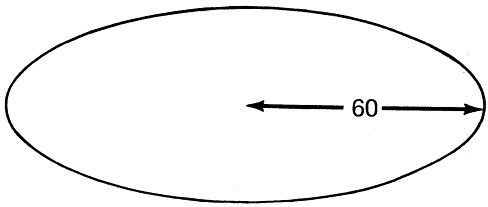


aspect affects the ratio of the x-radius to the y-radius. The default for *aspect* is 5/3 in low resolution, 5/6 in medium resolution and 5/12 in high resolution. These values give a visual circle assuming the standard screen aspect ratio of 4/3.

If *aspect* is less than one, then *r* is the x-radius. That is, the radius is measured in points in the horizontal direction. If *aspect* is greater than one, then *r* is the y-radius. For example,

```
10 SCREEN 1
20 CIRCLE (160,100),60,,,,5/18
```

will draw an ellipse like this:



In many cases, an *aspect* of 1 (one) will give nicer looking circles in medium resolution. This will also cause the circle to be drawn somewhat faster.

The last point referenced after a circle is drawn is the center of the circle.

Points that are off the screen are not drawn by CIRCLE.

Example: The following example draws a face.

CIRCLE
Statement

```
10 PI=3.141593
20 SCREEN 1 ' medium res. graphics
30 COLOR 0,1 ' black background, palette 1
40 'two circles in color 1 (cyan)
50 CIRCLE (120,50),10,1
60 CIRCLE (200,50),10,1
70 'two horizontal ellipses
80 CIRCLE (120,50),30,,,,5/18
90 CIRCLE (200,50),30,,,,5/18
100 'arc in color 2 (magenta)
110 CIRCLE (160,0),150,2, 1.3*PI, 1.7*PI
120 'arc, one side connected to center
130 CIRCLE (160,52),50,, 1.4*PI, -1.6*PI
```

Purpose: Sets all numeric variables to zero and all string variables to null. Options set the end of memory, the amount of stack space, and the size of video memory.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: CLEAR [,[*m*] [,*s*]] [,*v*]]

Remarks:

m is a byte count which, if specified, sets the maximum number of bytes for the BASIC workspace (where your program and data are stored, along with the interpreter work area). You would include *m* if you need to reserve space in storage for machine language programs.

s sets aside stack space for BASIC. The default is 512 bytes, or one-eighth of the available memory (whichever is smaller). You may want to include *s* if you use a lot of nested GOSUB statements or FOR–NEXT loops in your program, or if you use PAINT to do complex scenes.

v is an integer in the range of (2048 to 2^{24}) that specifies the total number of bytes to be set aside for video memory. The default is 16384. The integer (*v*) is used when you need to increase or decrease the size of your video memory. In order to execute a SCREEN 5 or SCREEN 6 statement, you must set aside

32K bytes for video memory. This requires a machine with 128K memory. This option is only supported in Cartridge BASIC.

CLEAR frees all memory used for data without erasing the program which is currently in memory. After a CLEAR, arrays are undefined; numeric variables have a value of zero; string variables have a null value; and any information set with any DEF statement is lost. (This includes DEF FN, DEF SEG, and DEF USR, as well as DEFINT, DEFDBL, DEFSNG, and DEFSTR.)

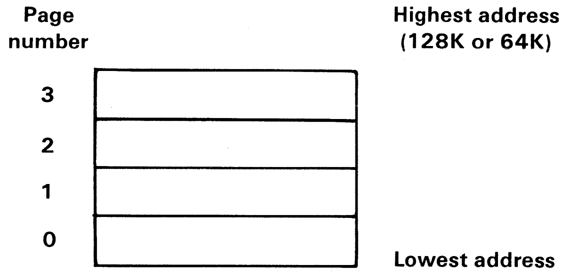
Executing a CLEAR command turns off any sound that is running and resets to Music Background. Also, PEN and STRIG are reset to OFF.

The last argument to CLEAR, *v*, is used with Cartridge BASIC only. It is used to increase and decrease video memory and to control the number of video pages available within the video memory. Each screen mode has a defined page size. The chart below illustrates:

SCREEN	PAGE SIZE
0 (40 wide)	2K
0 (80 wide)	4K
1	16K
2	16K
3	16K
4	16K
5	32K (requires 128K of machine storage)
6	32K (requires 128K of machine storage)

When you CLEAR to a specified video memory size, BASIC divides that area into the maximum number

of pages based on page size requirements. Below is a diagram of video memory, SCREEN 0, WIDTH 80, as it appears in the default size buffer of 16K:



Screen 0 - Width 80

Page numbering always begins with 0 and page 0 is always at the lowest address.

Referring to the chart, you can see that 16K may or may not be adequate storage for the screen mode you desire. For example, in the diagram above, BASIC has allocated four pages with the default 16K buffer. If you need to maintain only one visual page (refer to the SCREEN statement for a discussion of active and visual pages), you actually need only 4K of video memory. This would allow you to free 12K bytes of memory. The following CLEAR statement would do this for you:

```
CLEAR,,,4096
```

In 40 wide text mode, you could free a maximum of 14K leaving 2K for the video memory by executing:

```
CLEAR,,,2048
```

Since the page size for SCREEN 5 and SCREEN 6 is 32K, you must set aside at least 32K.

To be sure that you have allocated enough memory in the CLEAR statement for your application, use the formula below:

Total avail memory - allocated video memory =
Total avail memory- (size per page)*(no. of pages)

If you have more than one video page, it is possible to copy the contents of one page to another using the PCOPY statement (refer to the "PCOPY Statement" later in this chapter).

If you allocate less video memory than what is currently set aside, BASIC will put you into text mode, 40 wide, page 0. This mode has the smallest page size.

When video memory is expanded from 16K, it acquires space from the free memory space as designated in the memory map in Appendix I. This free memory area can also be referenced with the **/M** switch on the BASIC command line which sets the top of free memory. If at any time video memory and the top of the BASIC workspace meet, an **Out of memory** error will occur.

If you use CLEAR, it should be the first statement in your program because it erases all variables. For more information on paging, see the "SCREEN Statement" and the "PCOPY Statement" in this chapter.

The ERASE statement may be useful to free some memory without erasing all the data in the program. It erases only specified arrays from the work area. Refer to "ERASE Statement" in this chapter for details.

Example: This example clears all data from memory (without erasing the program):

```
CLEAR
```

The next example clears the data and sets the maximum workspace size to 32K bytes:

```
CLEAR,32768
```

The next example clears the data and sets the size of the stack to 2000 bytes:

```
CLEAR,,2000
```

The next example clears data, sets the maximum workspace for BASIC to 32K bytes, and sets the stack size to 2000:

```
CLEAR,32768,2000
```

The next example clears 32K bytes of video memory. This is required when using SCREEN 5 or SCREEN 6:

```
CLEAR,,,32768
```

The next example shows how you would determine the amount of memory you would need if you wanted to create a buffer of 4 pages of 16K each:

4 pages * 16K per page = 64K

and if you had 128K of memory, you could substitute for the formula the following:

$$128K - 64K = 128K - (16K * 4)$$
$$64K = 64K$$

The formula checks so you may execute the following:

```
CLEAR,,,65536
```

Purpose: Concludes I/O to a device or file.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: CLOSE [[#] *filenum* [,[#] *filenum*]...]

Remarks:

filenum is the number used on the OPEN statement.

The association between a particular file or device and its file number stops when CLOSE is executed. Subsequent I/O operations specifying that file number will be invalid. The file or device may be opened again using the same or a different file number; or the file number may be reused to open any device or file.

A CLOSE to a file or device opened for sequential output causes the final buffer to be written to the file or device.

A CLOSE with no file numbers specified causes all devices and files that have been opened to be closed.

Executing an END, NEW, RESET, SYSTEM or RUN without the **R** option causes all open files and devices to be automatically closed. STOP does not close any files or devices.

Refer also to "OPEN Statement" in this chapter for information about opening files.

Example:

CLOSE
Statement

```
100 CLOSE 1,#2,#3
```

Causes the files and devices associated with file numbers 1, 2, and 3 to be closed.

```
200 CLOSE
```

Causes all open devices and files to be closed.

Purpose: Clears the screen.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: CLS

Remarks: If the screen is in text mode, the active page (see "SCREEN Statement" in this chapter) is cleared to the background color (see "COLOR Statement," also in this chapter).

If the screen is in graphics mode (low or medium or high resolution), the entire screen memory is cleared to the background attribute 0.

The CLS statement also returns the cursor to the home position. In all modes, this means the cursor is located in the upper left-hand corner of the screen. In graphics mode, the graphics cursor is moved to the center of the screen ((80,100) in low resolution, (160,100) in medium resolution, (320,100) in high resolution). If view ports are in effect, the graphics cursor is moved to the center of the view port.

Changing the screen mode or width by using the SCREEN or WIDTH statements also clears the screen. The screen may also be cleared by pressing Ctrl-Fn/Home.

In Cartridge BASIC, when you are using the VIEW statement, CLS will only clear the active viewport. To clear the entire screen you must use VIEW to disable the viewport, and then use CLS to clear the screen.

CLS
Statement

Example:

```
5  SCREEN 0
10 COLOR 10,1
20 CLS
```

This example clears the screen to Blue.

Purpose: Sets the colors for the foreground, background, and border in text mode and in graphics modes, sets the background and palette or the foreground and background depending on the current screen mode.

The syntax of the COLOR statement depends on whether you are in text mode or graphics mode, as set by the SCREEN statement. When BASIC is first started, the foreground is white and the background and border are black.

In text mode, you can set the following:

SCREEN 0– Foreground, maximum attribute of 15
Character blink, if desired
Background, maximum attribute of 7
Border, maximum attribute of 15

In graphics mode, you can set the following:

SCREEN 1– Background, maximum attribute of 15 (0-15)
Palette, 1 of 2 palettes, 0 or 1
maximum attribute of 3 in each palette (0-3)

SCREEN 3– Foreground, maximum attribute of 15 (1-15)
Background, maximum color of 15 (0-15)

SCREEN 4– Foreground, maximum attribute of 3 (1-3)
Background, maximum color of 15 (0-15)

SCREEN 5- Foreground, maximum attribute of 15 (1-15)
Background, maximum color of 15 (0-15)

SCREEN 6- Foreground, maximum attribute of 3 (1-3)
Background, maximum color of 15 (0-15)

In graphics mode, the border is the same as the background color.

The COLOR Statement in Text Mode

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: COLOR [*foreground*] [,[*background*] [,*border*]]

Text mode only.

Remarks:

foreground is a numeric expression in the range of 0 to 31, for the character color.

background is a numeric expression in the range of 0 to 7 for the background color.

border is a numeric expression in the range of 0 to 15. It is the color for the border.

The following colors are allowed for *foreground*:

0	Black	8	Gray
1	Blue	9	Light Blue
2	Green	10	Light Green
3	Cyan	11	Light Cyan
4	Red	12	Light Red
5	Magenta	13	Light Magenta
6	Brown	14	Yellow
7	White	15	High-intensity White

Colors and intensity may vary depending on your display device.

You might like to think of colors 8 to 15 as "light" or "high-intensity" values of colors 0 to 7.

COLOR
Statement

You can make the characters blink by setting *foreground* equal to 16 plus the number of the desired attribute. That is, a value of 16 to 31 causes blinking characters.

You may select only colors 0 through 7 for *background.*

With the PALETTE and PALETTE USING statements, it is possible to obtain any color combination for foreground, background, and border.

Notes:

1. Foreground color may equal background color. This makes any character invisible. Changing the foreground or background color will make characters visible again.

2. Any parameter may be omitted. When parameters are not set, the old value is kept.

3. If the COLOR statement ends in a comma (,), you will get a **Missing operand** error, but the color will change. For example,

   ```
   COLOR ,7,
   ```

   is invalid.

4. Any values entered outside the range 0 to 255 result in an **Illegal function call** error. Previous values are kept.

Example:

```
10 COLOR 14,1,0
```

This sets a yellow foreground, a blue background, and a black border screen.

```
10 PRINT "Enter your ";
20 COLOR 15,0 'highlight next word
30 PRINT "password";
40 COLOR 7    'return to default (white/black)
50 PRINT " here: ";
60 COLOR 0    'invisible (black on black)
70 INPUT PASSWORD$
80 IF PASSWORD$="secret" THEN 120
90 ' blink and highlight error message
100 COLOR 31: PRINT "Wrong Password": COLOR 7
110 GOTO 10
120 COLOR 0,7  'reverse image (black on white)
130 PRINT "Program continues...";
140 COLOR 7,0  'return to default (white/black)
```

COLOR
Statement

The COLOR Statement in Graphics Mode

Versions:	Cassette	Cartridge	Compiler
	***	***	***

Graphics mode only.

Format: For Screen 1:

COLOR [*background*][,[*palette*]]

For Screens 3 to 6:

COLOR [*foreground]* [,[*background*]]

Remarks: **SCREEN 1:**

background is the attribute specifying the color of the background. It is an integer expression in the range 0 to 15.

palette is an integer expression which chooses one of two palettes. The maximum attribute in each palette is 3.

SCREENS 3-6:

foreground is the foreground attribute to be used for characters and graphic pixels. The foreground attribute is an integer expression in the range 1 to the maximum attribute allowed for the screen mode. The foreground attribute may not be zero. An **Illegal function call** error will result if *foreground* is 0.

background	is the color (*not* attribute) to be used for the background and border. *background* is an integer exppresion in the range of 0 to 15. This allows one of 16 possible colors for the background and border. These modes are only supported in Cartridge BASIC.

In SCREEN 1 the COLOR statement sets a background color and a palette of three colors. The colors selected when you choose each palette are as follows:

Color	Palette 0	Palette 1
1	Green	Cyan
2	Red	Magenta
3	Brown	White

You may select any one of these three attributes for display with the PSET, PRESET, LINE, CIRCLE, PAINT, VIEW, and DRAW statements. If *palette* is an even number, palette 0 is selected. This associates the colors Green, Red, and Brown to the attributes 1, 2, and 3. Palette 1 (Cyan, Magenta, White) is selected when *palette* is an odd number. This is also the default palette.

In SCREEN 4 and SCREEN 6, the default colors for attributes 1, 2 and 3 are cyan, magenta and white.

If through the PALETTE or PALETTE USING statements you change the color for attribute 0, your background will automatically change to this color since any change to attribute 0 will be regarded as a change to the background.

COLOR
Statement

COLOR for screen modes three through six provides much of the same flexibility of colors with text as COLOR in text mode does. With the PALETTE and PALETTE USING statements it is possible to choose any color for foreground and background for all modes.

Any values entered outside the range 0 to 255 cause an **Illegal function call** error. Previous values will be retained.

The COLOR statement has meaning only in those modes that use color. Attempting to use COLOR in screen 2 will result in an **Illegal function call** error.

Example:

```
5 SCREEN 1
10 COLOR 9,0
```

Sets the background to light blue, and selects palette 0.

```
20 COLOR ,1
```

The background stays light blue, and palette 1 is selected.

Purpose: Enables or disables trapping of communications activity to the specified communications port.

Versions:

Cassette	Cartridge	Compiler
	***	(**)

Format: COM(*n*) ON

COM(*n*) OFF

COM(*n*) STOP

Remarks:

n is the number of the communications port (1 or 2):

COM1: IBM Internal Modem or RS232 Serial port when IBM Modem is not present.

COM2: RS232 Serial port when IBM Modem is present.

A COM(*n*) ON statement must be executed to allow trapping by the ON COM(*n*) statement. After COM(*n*) ON, if a non-zero line number is specified in the ON COM(*n*) statement, BASIC checks to see if any characters have come in to the communications adapter every time a new statement is executed.

If COM(*n*) is OFF, no trapping takes place and any communication activity is not remembered even if it does take place.

If a COM(*n*) STOP statement has been executed, no trapping can take place. However, any communications activity that does take place is remembered so that an immediate trap occurs when COM(*n*) ON is executed.

Purpose: Passes variables to a chained program.

Note: This statement requires the use of DOS.

Versions:

Cassette	Cartridge	Compiler
	***	(**)

Format: COMMON *variable* [,*variable*]...

Remarks:

variable is the name of a variable that is to be passed to the chained-to program. Array variables are specified by appending "()" to the variable name.

The COMMON statement is used with the CHAIN statement. COMMON statements may appear anywhere in a program, although it is recommended that they appear at the beginning. Any number of COMMON statements may appear in a program, but the same variable cannot appear in more than one COMMON statement. If all variables are to be passed, use CHAIN with the ALL option and omit the COMMON statement.

Any arrays that are passed do not need to be dimensioned in the chained-to program.

Example: This example chains to program PROG3 on the diskette in drive A, and passes the array D along with the variables A, BEE1, C, and G$.

```
100 COMMON A,BEE1,C,D(),G$
110 CHAIN "A:PROG3"
```

CONT
Command

Purpose: Resumes program execution after a break.

Versions:

Cassette	Cartridge	Compiler
***	***	

Format: CONT

Remarks: The CONT command may be used to resume program execution after Fn/Break has been pressed, a STOP or END statement has been executed, or an error has occurred. Execution continues at the point where the break happened. If the break occurred after a prompt from an INPUT statement, execution continues with the reprinting of the prompt.

CONT is usually used with STOP for debugging. When execution is stopped, you can examine or change the values of variables using direct mode statements. You may then use CONT to resume execution, or you may use a direct mode GOTO, which resumes execution at a particular line number. CONT is invalid if the program has been edited during the break.

Example: In the following example, we create a long loop.

```
Ok
10 FOR A=1 TO 50
20 PRINT A;
30 NEXT A
RUN
 1  2  3  4  5  6  7  8  9  10  11 12
 13  14  15  16  17  18  19  20  21  22
 23  24  25  26  27  28  29
```

(At this point we interrupt the loop by pressing Fn/Break.)

```
.
.
.
Break in 20
Ok
CONT
 30  31  32  33  34  35  36  37  38  39
 40  41  42  43  44  45  46  47  48  49
 50
Ok
```

COS
Function

Purpose: Returns the trigonometric cosine function.

Versions: Cassette *** Cartridge *** Compiler ***

Format: *v* = COS(*x*)

Remarks:

x is the angle whose cosine is to be calculated.

The value of *x* must be in radians. To convert from degrees to radians, multiply the degrees by PI/180, where PI=3.141593.

The calculation of COS(*x*) is performed in single precision in Cassette BASIC and in either single or double precision in Cartridge BASIC.

Example: This example shows, first, that the cosine of PI radians is equal to -1. Then it calculates the cosine of 180 degrees by first converting the degrees to radians (180 degrees happens to be the same as PI radians).

```
Ok
10 PI=3.141593
20 PRINT COS(PI)
30 DEGREES=180
40 RADIANS=DEGREES*PI/180
50 PRINT COS(RADIANS)
RUN
-1
-1
Ok
```

Purpose: Converts *x* to a single-precision number.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: *v* = CSNG(*x*)

Remarks:

x is a numeric expression which will be converted to single-precision.

The rules outlined under "How BASIC Converts Numbers from One Precision to Another" in Chapter 3 are used for the conversion.

See the CINT and CDBL functions for converting numbers to the integer and double-precision data types.

Example:

```
Ok
10 A# = 975.3421222#
20 PRINT A#; CSNG(A#)
RUN
 975.3421222  975.3421
Ok
```

The value of the double-precision number A# is rounded at the seventh digit and returned as CSNG(A#).

CSRLIN
Variable

Purpose: Returns the vertical coordinate of the cursor.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: *v* = CSRLIN

Remarks: The CSRLIN variable returns the current line (row) position of the cursor on the active page. (The active page is explained under "SCREEN Statement" in this chapter.) The value returned will be in the range 1 to 25.

The POS function returns the column location of the cursor. Refer to "POS Function" in this chapter.

Refer to "LOCATE Statement" to see how to set the cursor line.

Example:

```
10 Y = CSRLIN   'record current line
20 X = POS(0)   'record current column
29 'print HI MOM on line 24
30 LOCATE 24,1: PRINT "HI MOM"
40 LOCATE Y,X   'restore position
```

This example saves the cursor coordinates in the variables X and Y, then moves the cursor to line 24 to put the words "HI MOM" on that line. Then the cursor is moved back to its old position.

CVI, CVS, CVD Functions

Purpose: Converts string variable types to numeric variable types.

Versions:

Cassette	Cartridge	Compiler
	***	***

Format: v = CVI(*2-byte string*)

v = CVS(*4-byte string*)

v = CVD(*8-byte string*)

Remarks: Numeric values that are read from a random file must be converted from strings into numbers. CVI converts a two-byte string to an integer. CVS converts a four-byte string to a single-precision number. CVD converts an eight-byte string to a double-precision number.

The CVI, CVS, and CVD functions do *not* change the bytes of the actual data. They only change the way BASIC interprets those bytes.

See also "MKI$, MKS$, MKD$ Functions" in this chapter, and Appendix B, "BASIC Diskette Input and Output."

CVI, CVS, CVD Functions

Example:

```
70 FIELD #1,4 AS N$, 12 AS B$
80 GET #1
90 Y=CVS(N$)
```

This example uses a random file (#1) which has fields defined as in line 70. Line 80 reads a record from the file. Line 90 uses the CVS function to interpret the first four bytes (N$) of the record as a single-precision number. N$ was probably originally a number which was written to the file using the MKS$ function.

Purpose: Stores the numeric and string constants that are read by the program's READ statement(s).

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: DATA *constant*[,*constant*]...

Remarks:

constant may be a numeric or string constant. No expressions are allowed in the list. The numeric constants may be in any format: integer, fixed point, floating point, hex, or octal. String constants in DATA statements do not need to be surrounded by quotation marks unless the string contains commas, colons, or significant leading or trailing blanks.

DATA statements may be placed anywhere in the program. A DATA statement may contain as many constants as will fit on a line. In addition, any number of DATA statements may be used in a program. The information contained in the DATA statements may be thought of as one continuous list of items, regardless of how many items are on a line or where the lines are placed in the program. The READ statements access the DATA statements in line number order.

The variable type (numeric or string) given in the READ statement must agree with the corresponding constant in the DATA statement or a **Syntax error** occurs.

DATA Statement

You cannot use the single quote (') to add comments to the end of a DATA statement. If you do, BASIC considers it part of a string. You may, however, use :REM to add a remark.

You can use the RESTORE statement to reread information from any line in the list of DATA statements. (See "RESTORE Statement" in this chapter.)

Example: See examples under "READ Statement" in this chapter.

Purpose: Sets or tells you the date.

> **Note:** This statement requires the use of DOS 2.10. If DOS 2.10 is not present, an **Illegal function call** error will occur.

Versions:

Cassette	Cartridge	Compiler
	***	***

Format: As a variable:

*v$* = DATE$

As a statement:

DATE$ = *x$*

Remarks: **For the variable (v$ = DATE$):**

A 10-character string of the form *mm-dd-yyyy* is returned. Here, *mm* represents two digits for the month, *dd* is the day of the month (also two digits), and *yyyy* is the year.

For the statement (DATE$ = x$):

x$* is a string expression which is used to set the current date. You may enter *x$ in any one of the following forms:

mm-dd-yy
mm/dd/yy
mm-dd-yyyy
mm/dd/yyyy

The year must be in the range 1980 to 2099. If you use only one digit for the month or day, a 0 (zero) is

added in front of it. If you give only one digit for the year, a zero is added to make it two digits. If you give only two digits for the year, the year is assumed to be 19*yy*.

Example:

```
Ok
10 DATE$= "8/29/82"
20 PRINT DATE$
RUN
08-29-1982
Ok
```

In the example, we set the date to August 29th, 1982. Notice how, when we read the date back using the DATE$ function, a zero was included in front of the month to make it two digits, and the year became 1982. Also, the month, day, and year are separated by hyphens even though we entered them as slashes.

Purpose: Defines and names a function that you write.

Versions:

Cassette	Cartridge	Compiler
***	***	(**)

Format: DEF FN*name*[(*arg* [,*arg*]...)] =*expression*

Remarks:

name is a valid variable name. This name, preceded by FN, becomes the name of the function.

arg is an argument. It is a variable name in the function definition that will be replaced with a value when the function is called. The arguments in the list represent, on a one-to-one basis, the values that are given when the function is called.

expression defines the returned value of the function. The type of the *expression* (numeric or string) must match the type declared by *name*.

The definition of the function is limited to one statement. Arguments (*arg*) that appear in the function definition serve only to define the function; they do not affect program variables that have the same name. A variable name used in the *expression* does not have to appear in the list of arguments. If it does, the value of the argument is supplied when the function is called. Otherwise, the current value of the variable is used.

DEF FN
Statement

The function type determines whether the function returns a numeric or string value. The type of the function is declared by *name*, in the same way as variables are declared (see "How to Declare Variable Types"in Chapter 3). If the type of *expression* (string or numeric) does not match the function type, a **Type mismatch** error occurs. If the function is numeric, the value of the expression is converted to the precision specified by *name* before it is returned to the calling statement.

A DEF FN statement must be executed to define a function before you may call that function. If a function is called before it has been defined, an **Undefined user function** error occurs. On the other hand, a function may be defined more than once. The most recently executed definition is used.

DEF FN is invalid in direct mode.

Example:

```
Ok
10 PI=3.141593
20 DEF FNAREA(R)=PI*R^2
30 INPUT "Radius? ",RADIUS
40 PRINT "Area is" FNAREA(RADIUS)
RUN
Radius?
```

(Suppose you respond with 2.)

```
Radius? 2
Area is 12.56637
Ok
```

Line 20 defines the function FNAREA, which calculates the area of a circle with radius R. The function is called in line 40.

Here is an example with two arguments:

```
Ok
10 DEF FNMUD(X,Y)=X-(INT(X/Y)*Y)
20 A = FNMUD(7.4,4)
30 PRINT A
RUN
 3.4
Ok
```

DEF SEG
Statement

Purpose: Defines the current "segment" of storage. A subsequent BLOAD, BSAVE, CALL, PEEK, POKE, or USR definition will define the actual physical address of its operation as an offset into this segment.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: DEF SEG [=*address*]

Remarks:

address is a numeric expression in the range 0 to 65535.

The initial setting for the segment when BASIC is started is BASIC's Data Segment (DS). BASIC's Data Segment is the beginning of your user workspace in memory. If you execute a DEF SEG statement which changes the segment, the value does *not* get reset to BASIC's DS when you issue a RUN command.

If *address* is omitted from the DEF SEG statement, the segment is set to BASIC's Data Segment.

If *address* is given, it should be a value based upon a 16 byte boundary. The value is shifted left 4 bits (multiplied by 16) to form the segment address for the subsequent operation. That is, if *address* is in hexadecimal, a 0 (zero) is added to get the actual segment address. BASIC does not perform any checking to assure that the segment value is valid.

DEF and SEG must be separated by a space. Otherwise, BASIC interprets the statement **DEFSEG=100** to mean: "assign the value 100 to the variable DEFSEG."

Any value entered outside the range indicated results in an **Illegal function call** error. The previous value will be retained.

Refer to Appendix C, "Machine Language Subroutines" for more information on using DEF SEG.

Example:

```
100 DEF SEG ' restore segment to BASIC DS
200 ' set segment to color screen buffer
210 DEF SEG=&HB800
```

In the second example, the screen buffer is at absolute address B8000 hex. Since segments are specified on 16 byte boundaries, the last hex digit is dropped on the DEF SEG specification.

DEFtype Statements

Purpose: Declares variable types as integer, single-precision, double-precision, or string variable types.

Versions:

Cassette	Cartridge	Compiler
***	***	(**)

Format: DEF*type letter*[-*letter*] [,*letter* [-*letter*]]...

Remarks:

type is INT, SNG, DBL, or STR.

letter is a letter of the alphabet (A-Z).

A DEF*type* statement declares that the variable names beginning with the letter or letters specified will be that type of variable. However, a type declaration character (%, !, #, or $) always takes precedence over a DEF*type* statement in the typing of a variable. Refer to "How to Declare Variable Types" in Chapter 3.

If no type declaration statements are encountered, BASIC assumes that all variables without declaration characters are single-precision variables.
If type declaration statements are used, they should be at the beginning of the program. The DEF*type* statement must be executed before you use any variables which it declares.

Example:

```
Ok
10 DEFDBL L-P
20 DEFSTR A
30 DEFINT X,D-H
40 ORDER = 1#/3: PRINT ORDER
50 ANIMAL = "CAT": PRINT ANIMAL
60 X=10/3: PRINT X
RUN
 .3333333333333333
CAT
 3
Ok
```

Line 10 declares that all variables beginning with the letter L, M, N, O, or P will be double-precision variables.

Line 20 causes all variables beginning with the letter A to be string variables.

Line 30 declares that all variables beginning with the letter X, D, E, F, G, or H will be integer variables.

DEF USR Statement

Purpose: Specifies the starting address of a machine language subroutine, which is later called by the USR function.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: DEF USR[*n*]=*offset*

Remarks:

n may be any digit from 0 to 9. It identifies the number of the USR routine whose address is being specified. If *n* is omitted, DEF USR0 is assumed.

offset is an integer expression in the range 0 to 65535. The value of *offset* is added to the current segment value to obtain the actual starting address of the USR routine. See "DEF SEG Statement" in this chapter.

It is possible to redefine the address for a USR routine. Any number of DEF USR statements may appear in a program, thus allowing access to as many subroutines as necessary. The most recently executed value is used for the offset.

Refer to Appendix C, "Machine Language Subroutines" for complete information.

Example:

```
200 DEF SEG= &H1000
210 DEF USR0= 2400
500 X=USR0(Y+2)
```

This example calls a routine at absolute location 24000 in memory.

DELETE Command

Purpose: Deletes program lines.

Versions:

Cassette	Cartridge	Compiler
***	***	

Format: DELETE [*line1*] [-*line2*]

DELETE [*line1*-]

Remarks:

line1 is the line number of the first line to be deleted.

line2 is the line number of the last line to be deleted.

The DELETE command erases the specified range of lines from the program. BASIC always returns to command level after a DELETE is executed.

DELETE *line1*- deletes all lines from the specified line number through the end of the program. This is for Cartridge BASIC only.

A period (.) may be used in place of the line number to indicate the current line. If you specify a line number that does not exist in the program, an **Illegal function call** error occurs.

Examples: This example deletes line 40:

```
DELETE 40
```

This example deletes lines 40 through 100, inclusive:

```
DELETE 40-100
```

This example deletes line 40 through the end of the program:

```
DELETE 40-
```

The last example deletes all lines up to and including line 40:

```
DELETE -40
```

DIM
Statement

Purpose: Assigns the maximum values for array variable subscripts and makes space available for them.

Versions:

Cassette	Cartridge	Compiler
***	***	(**)

Format: DIM *variable*(*subscripts*) [,*variable*(*subscripts*)]...

Remarks:

variable is the name used for the array.

subscripts is a list of numeric expressions, separated by commas, which define the dimensions of the array.

When entered, the DIM statement first sets all the elements of the specified numeric arrays to zero. String array elements may vary in length, and begin with a null value (zero length).

If an array variable name is used without a DIM statement, the maximum value its subscript can have is 10. If a subscript of greater than 10 is used, a **Subscript out of range** error occurs.

The minimum value for a subscript is always 0, the OPTION BASE statement says otherwise (see "OPTION BASE Statement" in this chapter). The maximum number of dimensions for an array is 255. The maximum number of elements per dimension is 32767. Both of these numbers are also limited by memory size and statement length.

If you try to define an array more than once, a **Duplicate Definition** error occurs. You may use the ERASE statement to erase an array. For more information about arrays, see "Arrays"in Chapter 3.

Example:

```
Ok
10 WRRMAX=2
20 DIM SIS(12), WRR$(WRRMAX,2)
30 DATA 26.5, 37, 8,29,80, 9.9, &H800
40 DATA 7, 18, 55, 12, 5, 43
50 FOR I=0 TO 12
60 READ SIS(I)
70 NEXT I
80 DATA SHERRY, ROBERT, "A:"
90 DATA "HI, SCOTT", HELLO, GOOD-BYE
100 DATA BOCA RATON, DELRAY, MIAMI
110 FOR I=0 TO 2: FOR J=0 TO 2
120 READ WRR$(I,J)
130 NEXT J,I
140 PRINT SIS(3); WRR$(2,0)
RUN
 29 BOCA RATON
Ok
```

This example creates two arrays: a one-dimensional numeric array named SIS with 13 elements, SIS(0) through SIS(12); and a two-dimensional string array named WRR$, with three rows and three columns.

DRAW Statement

Purpose: Draws an object as specified by *string*.

Versions:

Cassette	Cartridge	Compiler
	***	(**)

Graphics mode only.

Format: DRAW *string*

Remarks: You use the DRAW statement with a *graphics definition language* to draw. The language commands are contained in the string expression *string*. The string defines an object that is drawn when BASIC executes the DRAW statement. During execution, BASIC examines the value of *string* and interprets single letter commands from the string. These commands are detailed below.

The following movement commands begin movement from the graphic cursor. After each command, the graphic cursor is the last point the command draws.

U n Move up.

D n Move down.

L n Move left.

R n Move right.

E n Move diagonally up and right.

F n Move diagonally down and right.

G n Move diagonally down and left.

H n Move diagonally up and left.

n in each of the preceding commands indicates the distance to move. The number of points moved is *n* times the scaling factor (set by the **S** command).

M x,y Move absolute or relative. If *x* has a plus sign (+) or a minus sign (-) in front of it, it is relative. Otherwise, it is absolute.

The graphic statement DRAW does not take into account the aspect ratio of the current screen mode. That is, DRAW "R50 U50" will plot exactly 50 points to the right and then 50 up. For more information see "Graphics Mode" in chapter 3.

The following two prefix commands may precede any of the above movement commands.

B Move, but don't plot any points.

N Move, but return to the original position when finished.

The following commands are also available:

A n Set angle *n*. *n* may range from 0 to 3, where 0 is 0 degrees, 1 is 90, 2 is 180, and 3 is 270. Figures rotated 90 or 270 degrees are scaled so that they appear the same size as with 0 or 180 degrees on a display screen with standard aspect ratio 4/3.

TA n Turn angle *n*. *n* can range from -360 to +360. If *n* is positive (+), the angle turns counterclockwise. If *n* is negative (-), the angle turns clockwise. Values entered that are outside of the range -360 to +360 cause an **Illegal function call** error.

C n Set color *n*. *n* may range from 0 to 15 in low resolution, 0 to 3 or 0 to 15 in medium resolution, and 0 to 1 or 0 to 3 in high resolution. *n* selects the attribute from the current palette and 0 is the background attribute. The default attribute is always the maximum attribute for the current screen mode. In high resolution, SCREEN 2, *n* equal to 0 (zero) indicates black, and the default of 1 (one) indicates white.

S n Set scale factor. *n* may range from 1 to 255. *n* divided by 4 is the scale factor. For example, if *n*=1, then the scale factor is 1/4. The scale factor multiplied by the distances given with the U, D, L, R, E, F, G, H, and relative M commands gives the actual distance moved. The default value is 4, so the scale factor is 1.

X variable;

Execute substring. This allows you to execute a second string from within a string.

P paint,boundary

Set figure color to *paint* and edge attribute to *boundary*. The *paint* parameter can range from 0 to 3 or 0 to 15, depending on the current screen mode. In low resolution there are 16 attributes available (0 to 15) In medium resolution, there are 4 (0 to 3) or 16 (0 to 15) attributes available, depending on the current screen mode. In high resolution there are 2 (0 to 1) or 4 (0 to 3) attributes available, depending on the current screen mode. The default attribute is always the

maximum attribute for the current screen mode. 0 is the background attribute. For more information see "Graphics Modes" in chapter 3. The *boundary* parameter is the edge attribute of the figure to be filled in, in the range 0 to 15 as described in *paint*. You must specify both *paint* and *boundary* or an error will occur. This command does not support tile painting.

In all these commands, the *n, x*, or *y* argument can be a constant like **123** or it can be **=*variable*;** where *variable* is the name of a numeric variable. The semicolon (;) is required when you use a variable this way, or in the **X** command. Otherwise, a semicolon is optional between commands. Spaces are ignored in *string*. For example, you could use variables in a move command this way: **M+=X1;, -=X2;**

You can also specify variables in the form VARPTR$(*variable*), instead of **=*variable*;**. This is the only form that can be used in compiled programs. For example:

One Method	**Alternative Method**
DRAW "XA$;"	DRAW "X"+VARPTR$(A$)
DRAW "S=SC;"	DRAW "S="+VARPTR$(SC)

The **X** command can be a very useful part of DRAW. By using it, you can define a part of an object separate from the entire object. For example, a leg could be part of a man. You can also use **X** to draw a string of commands more than 255 characters long.

Aspect Ratio

The Aspect Ratio is used to correct the shape of objects drawn on a non-linear surface. The idea is to be able to draw a square, for example, that indeed looks square.

If there were 640 by 640 dots on a screen evenly spaced along the x and y axis, then we would say that the aspect ratio is "1 to 1" or 1/1. This is an ideal surface. If we execute the statement:

```
DRAW "R100 D100 L100 U100"
```

Then the box would appear very square.

However, this is not the case. BASIC supports three screen resolutions, each with their own aspect ratio. These are:

Resolution		**Aspect Ratio**
Low resolution	160 by 200 dots	5/3
Medium resolution	320 by 200 dots	5/6
High resolution	640 by 200 dots	5/12

Note that 160/200 is not 5/3. This is because the spacing between dots is different along the x axis than it is along the y axis.

In order to draw a box that appears square in any of the above resolutions, scale the y axis by the corresponding aspect ration, or scale the x axis by 1/aspect ratio (3/5 for low resolution).

For example, to draw a square box 100 high, scale the x axis as follows:

```
REM 100*6/5 is 120
DRAW "R120 D100 L120 U100"
```

To draw a square box 100 wide, scale the y axis as follows:

```
REM 100*5/6 is 83
DRAW "R100 D83 L100 U83"
```

DRAW Statement

Examples: To draw a box:

```
5 SCREEN 1
10 A=20
20 DRAW "U=A;R=A;D=A;L=A;"
```

To draw a box and paint the interior:

```
10 DRAW "U50R50D50L50" 'Draw a box
20 DRAW "BE10" 'Move up and right into box
30 DRAW "P1,3" 'Paint interior
```

To draw a triangle:

```
10 SCREEN 1
20 DRAW "E15 F15 L30"
```

To create a "shooting star":

```
10 SCREEN 1,0: COLOR 0,0: CLS
20 DRAW "BM300,25" ' initial point
30 STAR$="M+7,17 M-17,-12 M+20,0 M-17,12 M+7,-17"
40 FOR SCALE=1 TO 40 STEP 2
50 DRAW "C1;S=SCALE; BM-2,0;XSTAR$;"
60 NEXT
```

To draw some "spokes":

```
10 FOR D=0 to 360 STEP 10
20 DRAW "TA=D; NU50"
30 NEXT D
```

Purpose: Displays a line for editing.

Versions:

Cassette	Cartridge	Compiler
***	***	

Format: EDIT *line*

Remarks:

line is the line number of a line existing in the program. If there is no such line, an **Undefined line number** error occurs. A period (.) can be used for the line number to refer to the current line.

The EDIT statement displays the line specified and positions the cursor under the first digit of the line number. The line may then be changed as described under "The BASIC Program Editor" in Chapter 2.

A period (.) can be used for the line number to refer to the current line. For example, if you have just entered a line and want to change it, the command **EDIT** redisplays the line for editing.

LIST also displays program lines for changing. Refer to "LIST Command" in this chapter.

END Statement

Purpose: Terminates program execution, closes all files, and returns to command level.

Versions:

Cassette	Cartridge	Compiler
***	***	(**)

Format: END

Remarks: END statements may be placed anywhere in the program to terminate execution. END is different from STOP in two ways:

- END does not cause a **Break** message to be printed.
- END closes all files.

An END statement at the end of a program is optional. BASIC always returns to command level after an END is executed.

Example:

```
520 IF K>1000 THEN END ELSE GOTO 20
```

This example ends the program if K is greater than 1000. Otherwise, the program branches to line number 20.

EOF Function

Purpose: Indicates an end of file condition.

Versions:

Cassette	Cartridge	Compiler
***	***	(**)

Format: *v* = EOF(*filenum*)

Remarks:

filenum is the number specified on the OPEN statement.

The EOF function is useful for avoiding an **Input past end** error. EOF returns -1 (true) if end of file has been reached on the specified file. A 0 (zero) is returned if end of file has not been reached.

EOF is significant only for a file opened for sequential input from diskette or cassette, or for a communications file. A -1 for a communications file means that the buffer is empty.

In Cartridge BASIC, EOF(0) returns the end of file condition on standard input devices used with redirection of I/O.

Example: This example reads information from the sequential file named "DATA." Values are read into the array M until end of file is reached.

```
10 C=0: OPEN "DATA" FOR INPUT AS #1
20 IF EOF(1) THEN END
30 INPUT #1,M(C)
40 C=C+1: GOTO 20
```

ERASE Statement

Purpose: Eliminates arrays from a program.

Versions:

Cassette	Cartridge	Compiler
***	***	

Format: ERASE *arrayname*[,*arrayname*]...

Remarks:

arrayname is the name of an array you want to erase.

You might want to use the ERASE statement if you are running short of storage space while running your program. After arrays are erased, the space in memory which had been allocated for the arrays may be used for other purposes.

ERASE can also be used when you want to redimension arrays in your program. If you try to redimension an array without first erasing it, a **Duplicate Definition** error occurs.

The CLEAR command erases *all* variables from the work area.

Example:

```
Ok
10 START=FRE("")
20 DIM BIG(100,100)
30 MIDDLE=FRE("")
40 ERASE BIG
50 DIM BIG(10,10)
60 FINAL=FRE("")
70 PRINT START, MIDDLE, FINAL
RUN
 62808          21980          62289
Ok
```

This example uses the FRE function to show how ERASE can be used to free memory. The array BIG used up about 40K bytes of memory (62808-21980) when it was dimensioned as BIG(100,100). After it was erased, it could be redimensioned to BIG(10,10), and it only took up a little more than 500 bytes (62808-62289).

The actual values returned by the FRE function may be different on your computer.

ERR and ERL Variables

Purpose: Return the error code and line number associated with an error.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: *v* = ERR

v = ERL

Remarks: The variable ERR contains the error code for the last error, and the variable ERL contains the line number of the line in which the error was detected. The ERR and ERL variables are usually used in IF and THEN statements to direct program flow in the error handling routine (refer to "ON ERROR Statement" in this chapter).

If you do test ERL in an IF–THEN statement, be sure to put the line number on the right side of the relational operator, like this:

```
IF ERL = line number THEN ...
```

The number must be on the right side of the operator for it to be renumbered by RENUM.

If the statement that caused the error was a direct mode statement, ERL will contain 65535. Since you do not want this number to be changed during a RENUM, if you want to test whether an error occurred in a direct mode statement you should use the form:

```
IF 65535 = ERL THEN ...
```

ERR and ERL can be set using the ERROR statement (see next section).

BASIC error codes are listed in Appendix A, "Messages."

Example:

```
10 ON ERROR GOTO 100
20 LPRINT "This goes to the printer"
30 END
100 IF ERR=27 THEN LOCATE 23,1:
    PRINT "Check printer": RESUME
```

This example tests for a common problem: forgetting to put paper in the printer, or forgetting to switch it on.

ERROR Statement

Purpose: Simulates the occurrence of a BASIC error; or allows you to define your own error codes.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: ERROR *n*

Remarks:

n must be an integer expression between 0 and 255.

If the value of *n* is the same as an error code used by BASIC (see Appendix A, "Messages"), the ERROR statement simulates the occurrence of that error. If an error handling routine has been defined by the ON ERROR statement, the error routine is entered. Otherwise the error message corresponding to the code is displayed, and execution halts. (See first example below.)

To define your own error code, use a value that is different from any used by BASIC. (We suggest you use the highest available values; for example, values greater than 200.) This new error code may then be tested in an error handling routine, just like any other error. (See second example below.)

If you define your own code in this way, and you don't handle it in an error handling routine, BASIC displays the message **Unprintable error**, and execution halts.

Example: The first example simulates a **String too long** error.

```
Ok
10 T = 15
20 ERROR T
RUN
String too long in line 20
Ok
```

The next example is a part of a game program that allows you to make bets. By using an error code of 210, which BASIC doesn't use, the program traps the error if you exceed the house limit.

```
110 ON ERROR GOTO 400
120 INPUT "WHAT IS YOUR BET";B
130 IF B > 5000 THEN ERROR 210
.
.
.
400 IF ERR = 210 THEN PRINT "HOUSE LIMIT IS $5000"
410 IF ERL = 130 THEN RESUME 120
```

EXP
Function

Purpose: Calculates the exponential function.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: *v* = EXP(*x*)

Remarks: *x* may be any numeric expression.

This function returns the mathematical number *e* raised to the *x* power. *e* is the base for natural logarithms. An overflow occurs if *x* is greater than 88.02969.

EXP(*x*) is calculated in single precision in Cassette BASIC and in either single or double precision in Cartridge BASIC.

Example:

```
Ok
10 X = 2
20 PRINT EXP(X-1)
RUN
 2.718282
Ok
```

This example calculates *e* raised to the (2-1) power, which is simply *e*.

Purpose: Allocates space for variables in a random file buffer. Note: This statement requires the use of DOS 2.10.

Versions:

Cassette	Cartridge	Compiler
	***	***

Format: FIELD [#]*filenum, width* AS *stringvar* [,*width* AS *stringvar*]...

Remarks:

filenum is the number under which the file was opened.

width is a numeric expression specifying the number of character positions to be allocated to *stringvar.*

stringvar is a string variable which will be used for random file access.

A FIELD statement defines variables that are used to get data out of a random buffer after a GET or to enter data into the buffer for a PUT.

The statement:

```
FIELD 1, 20 AS N$, 10 AS ID$, 40 AS ADD$
```

allocates the first 20 positions (bytes) in the random file buffer to the string variable N$, the next 10 positions to ID$, and the next 40 positions to ADD$. FIELD does *not* actually place any data into the random file buffer. This is done by the LSET and RSET statements (see "LSET and RSET Statements" in this chapter).

FIELD does not "remove" data from the file either. Information is read from the file into the random file buffer with the GET (file) statement. Information is read from the buffer by simply referring to the variables defined in the FIELD statement.

The total number of bytes allocated in a FIELD statement must not exceed the record length that was specified when the file was opened. Otherwise, a **Field overflow** error occurs.

Any number of FIELD statements may be executed for the same file number, and all FIELD statements that have been executed are in effect at the same time. Each new FIELD statement redefines the buffer from the first character position, so this has the effect of having multiple field definitions for the same data.

> **Note:** Be careful about using a fielded variable name in an input or assignment statement. Once a variable name is defined in a FIELD statement, it points to the correct place in the random file buffer. If a subsequent input statement or LET statement with that variable name on the left side of the equal sign is executed, the variable is moved to string space and is no longer in the file buffer.

See Appendix B, "BASIC Diskette Input and Output" for a complete explanation of how to use random files.

Example:

```
10 OPEN "A:CUST" AS #1
20 FIELD 1, 2 AS CUSTNO$, 30 AS CUSTNAME$,
     35 AS ADDR$
30 LSET CUSTNAME$="O'NEIL INC"
40 LSET ADDR$="50 SE 12TH ST, NY, NY"
50 LSET CUSTNO$=MKI$(7850)
60 PUT 1,1
70 GET 1,1
80 CNUM%= CVI(CUSTNO$): N$ = CUSTNAME$
90 PRINT CNUM%, N$, ADDR$
```

This example opens a file named "CUST" as a random file. The variable CUSTNO$ is assigned to the first 2 positions in each record, CUSTNAME$ is assigned to the next 30 positions, and ADDR$ is assigned to the next 35 positions. Lines 30 through 50 put information into the buffer, and the PUT statement in line 60 writes the buffer to the file. Line 70 reads back that same record, and line 90 displays the three fields. Note in line 80 that it is okay to use a variable name which was defined in a FIELD statement on the *right* side of an assignment statement.

FILES
Command

Purpose: Displays the names of files residing on the current directory of a diskette. The FILES command in BASIC is similar to the DIR command in DOS.

> **Note:** This command requires the use of DOS 2.10. If DOS 2.10 is not present, an Illegal function call error will occur.

Versions:	Cassette	Cartridge	Compiler
		***	(**)

Format: FILES [*filespec*]

Remarks:

filespec is a string expression for the file specification as explained under "Naming Files" in Chapter 3. If *filespec* is omitted, all the files on the current directory of the DOS default drive will be listed.

All files matching the filename are displayed. The filename may contain question marks (?). A question mark matches any character in the name or extension. An asterisk (*) as the first character of the name or extension will match any name or any extension.

If a drive is specified as part of *filespec*, files which match the specified filename on the current directory of that drive are listed. Otherwise, the DOS default drive is used.

Example:

```
FILES
```

This displays all files on the current directory of the DOS default drive.

```
FILES "*.BAS"
```

This displays all files with an extension of **.BAS** on the current directory of the DOS default drive.

```
FILES "TEST??.BAS"
```

This lists each file on the current directory of the DOS default drive that has a filename beginning with TEST followed by up to two other characters, and an extension of .BAS .

In addition to listing all the files on the current directory of the drive, BASIC also displays the current directory name and the number of bytes free.

When using tree-structured directories in Cartridge BASIC, each sub-directory contains two special entries—you will see them listed when you use the FILES command to list a sub-directory. The first contains a single period instead of a filename. It identifies this "file" as a sub-directory. The second entry contains two periods instead of a filename, and is used to locate the higher level directory that defines this sub-directory (the "parent" of the sub-directory).

```
FILES
A:\LEVEL1

        .   <DIR>        ..  <DIR>

 32824 Bytes free
```

This example lists all files in the current sub-directory called LEVEL1 on drive A.

FILES
Command

```
FILES "LVL1\"
```

The FILES command can also be used to list files in other directories. The example above lists all files in the sub-directory LVL1. The backslash must be used after the directory name.

```
FILES "LVL2\*.BAS"
```

This example lists all files in the directory LVL2 with an extension of .BAS.

Purpose: Truncates *x* to an integer.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: *v* = FIX(*x*)

Remarks: *x* may be any numeric expression.

FIX strips all digits to the right of the decimal point and returns the value of the digits to the left of the decimal point.

The difference between FIX and INT is that FIX does not return the next lower number when *x* is negative.

See the INT and CINT functions, which also return integers.

Example:

```
Ok
PRINT FIX(45.67)
 45
Ok
PRINT FIX(-2.89)
-2
Ok
```

Note in the examples how FIX does *not* round the decimal part when it converts to an integer.

FOR and NEXT Statements

Purpose: Performs a series of instructions in a loop a given number of times.

Versions:

Cassette	Cartridge	Compiler
***	***	(**)

Format: FOR *variable*=*x* TO *y* [STEP *z*]

.

.

.

NEXT [*variable* [,*variable*]...]

Remarks:

variable is an integer or single-precision variable to be used as a counter.

x is a numeric expression which is the initial value of the counter.

y is a numeric expression which is the final value of the counter.

z is a numeric expression to be used as an increment.

The program lines following the FOR statement are executed until the NEXT statement is encountered. Then the counter is incremented by the amount specified by the STEP value (*z*). If you do not specify a value for *z*, the increment is assumed to be 1 (one). A check is performed to see if the value of the counter is now greater than the final value *y*. If it is not greater, BASIC branches back to the statement after the FOR statement and the process is

repeated. If it is greater, execution continues with the statement following the NEXT statement. This is a FOR–NEXT loop.

If the value of *z* is negative, the test is reversed. The counter is decremented each time through the loop, and the loop is executed until the counter is less than the final value.

The body of the loop is skipped if *x* is already greater than *y* when the STEP value is positive, or if *x* is less than *y* when the STEP value is negative. If *z* is zero, an infinite loop is created unless you provide some way to set the counter greater than the final value.

Program performance will be improved if you use integer counters whenever possible.

Nested Loops

FOR–NEXT loops may be nested; that is, one FOR–NEXT loop may be placed inside another FOR–NEXT loop. When loops are nested, each loop must have a unique variable name as its counter. The NEXT statement for the inside loop must appear before that for the outside loop. If nested loops have the same end point, a single NEXT statement may be used for all of them.

FOR and NEXT Statements

A NEXT statement of the form:

```
NEXT var1, var2, var3 ...
```

is equivalent to the sequence of statements:

```
NEXT var1
NEXT var2
NEXT var3
.
.
.
```

The variable(s) in the NEXT statement may be omitted, in which case the NEXT statement matches the most recent FOR statement. If you are using nested FOR–NEXT loops, you should include the variable(s) on all the NEXT statements. It is a good idea to include the variables to avoid confusion; but it can be necessary if you do any branching out of nested loops. (However, using variable names on the NEXT statements will cause your program to execute somewhat slower.)

If a NEXT statement is encountered before its corresponding FOR statement, a **NEXT without FOR** error occurs.

Example: The first example shows a FOR–NEXT loop with a STEP value of 2.

```
Ok
10 J=10: K=30
20 FOR I=1 TO J STEP 2
30 PRINT I;
40 K=K+10
50 PRINT K
60 NEXT
RUN
 1  40
 3  50
 5  60
 7  70
 9  80
Ok
```

In this example, the loop does not execute because the initial value of the loop is more than the final value:

```
Ok
10 J=0
20 FOR I=1 TO J
30 PRINT I
40 NEXT I
RUN
Ok
```

This next program will result in a **NEXT without FOR** error. There may be only one NEXT statement for every FOR statement. (This is different from other versions of BASIC which allow a different physical NEXT statement when jumping out of a loop.)

```
10 FOR I=1 TO 5
20 IF I=2 GOTO 50
30 NEXT
40 GOTO 60
50 NEXT
60 END
```

FOR and NEXT Statements

In the last example, the loop executes ten times. The final value for the loop variable is always set before the initial value is set. (This is different from some other versions of BASIC, which set the initial value of the counter before setting the final value. In another BASIC the loop in this example might execute six times.)

```
Ok
10 I=5
20 FOR I=1 TO I+5
30 PRINT I;
40 NEXT
RUN
 1  2  3  4  5  6  7  8  9  10
Ok
```

Purpose: Returns the number of bytes in memory that are not being used by BASIC. This number does not include the size of the reserved portion of the interpreter work area (normally 2.5K to 4K bytes).

Versions:

Cassette	Cartridge	Compiler
***	***	(**)

Format: *v* = FRE(*x*)

v = FRE(*x$*)

Remarks: *x* and *x$* are dummy arguments.

Since strings in BASIC can have variable lengths (each time you do an assignment to a string its length may change), strings are manipulated dynamically. For this reason, string space may become fragmented.

FRE with any string value causes a housecleaning before returning the number of free bytes. *Housecleaning* is when BASIC collects all its useful data and frees up unused areas of memory that were once used for strings. The data is compressed so you can continue until you really run out of space.

BASIC also automatically does a housecleaning when it is running out of usable work area. Be patient, housecleaning may take a while.

CLEAR ,*n* sets the maximum number of bytes for the BASIC workspace. FRE returns the amount of free storage in the BASIC workspace. If nothing is in the workspace, then the value returned by FRE

will be 2.5K to 4K bytes (the size of the reserved interpreter work area) smaller than the number of bytes set by CLEAR.

Example:

```
Ok
PRINT FRE(0)
 14542
Ok
```

The actual value returned by FRE on your computer may differ from this example.

Purpose: Reads a record from a random file into a random buffer.

> **Note:** This statement requires the use of DOS 2.10.

Versions:

Cassette	Cartridge	Compiler
	***	***

Format: GET [#]*filenum*[, *number*]

Remarks:

filenum is the number under which the file was opened.

number is the number of the record to be read, in the range 16,777,215. If *number* is omitted, the next record (after the last GET) is read into the buffer.

After a GET statement, INPUT #, LINE INPUT #, or references to variables defined in the FIELD statement may be used to read characters from the random file buffer. Refer to Appendix B, "BASIC Diskette Input and Output," more complete information on using GET.

Because BASIC and DOS block as many records as possible in 512 byte sectors, the GET statement does not necessarily perform a physical read from the diskette.

GET may also be used for communications files. In this case *number* is the number of bytes to read from

the communications buffer. This number cannot exceed the value set by the LEN option on the OPEN "COM... statement.

The maximum record number allowed in the BASIC Compiler is 32767 instead of 16,777,215.

Example:

```
10 OPEN "A:CUST" AS #1
20 FIELD 1, 30 AS CUSTNAME$, 30 AS ADDR$,
       35 AS CITY$
30 GET 1
40 PRINT CUSTNAME$, ADDR$, CITY$
```

This example opens the file "CUST" for random access, with fields defined in line 20. The GET statement on line 30 reads a record into the file buffer. Line 40 displays the information from the record that was read.

Purpose: Reads points from an area of the screen.

Versions:

Cassette	Cartridge	Compiler
	***	***

Graphics mode only.

Format: GET (*x1,y1*)-(*x2,y2*),*arrayname*

Remarks:

(x1,y1), (x2,y2)
are coordinates in either absolute or relative form. Refer to "Specifying Coordinates" under "Graphics Modes" in Chapter 3 for information on coordinates.

arrayname is the name of the array you want to hold the information.

GET reads the colors of the points within the specified rectangle into the array. The specified rectangle has points (*x1,y1*) and (*x2,y2*) as opposite corners. (This is the same as the rectangle drawn by the LINE statement using the **B** option.)

GET and PUT can be used for high speed object motion in graphics mode. You might think of GET and PUT as "bit pump" operations which move bits onto (PUT) and off of (GET) the screen. Remember that PUT and GET are also used for random access files, but the syntax of these statements is different.

The array is used simply as a place to hold the image and must be numeric; it may be any precision, however. The required size of the array, in bytes, is:

4+INT((*x***bitsperpixel*+7)/8)**y*

where *x* and *y* are the lengths of the horizontal and vertical sides of the rectangle, respectively. The value of *bitsperpixel* is 4 in low resolution, 2 or 4 in medium resolution, and 1 or 2 in high resolution depending upon the current screen mode.

For example, suppose we want to use the GET statement to get a 10 by 12 image in medium resolution. The number of bytes required is 4+INT((10*2+7)/8)*12, or 40 bytes. The bytes per element of an array are:

- 2 for integer string
- 4 for single-precision string
- 8 for double-precision string

Therefore, we could use an integer array with at least 20 elements.

The information from the screen is stored in the array as follows:

1. two bytes giving the x dimension in bits
2. two bytes giving the y dimension in bits
3. the data itself

It is possible to examine the x and y dimensions and even the data itself if an integer array is used. The x dimension is in element 0 of the array, and the y dimension is in element 1. Keep in mind, however, that integers are stored low byte first, then high byte; but the data is actually transferred high byte first, then low byte.

The data for each row of points in the rectangle is left justified on a byte boundary, so if there are less than a multiple of eight bits stored, the rest of the byte will be filled with zeros.

PUT and GET work significantly faster in all resolutions when *x1* MOD (8/*bitsperpixel*) is equal to zero. This is a special case where the rectangle boundaries fall on the byte boundaries.

GOSUB and RETURN Statements

Purpose: Branches to and returns from a subroutine.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: GOSUB *line*

.

.

.

RETURN

Remarks:

line is the line number of the first line of the subroutine.

A subroutine may be called any number of times in a program, and a subroutine may be called from within another subroutine. Such nesting of subroutines is limited only by available memory.

The RETURN statement causes BASIC to branch back to the statement following the most recent GOSUB statement. A subroutine may contain more than one RETURN statement, if you want to return from different points in the subroutine. Subroutines may appear anywhere in the program.

To prevent your program from accidentally entering a subroutine, you may want to put a STOP, END, or GOTO statement before the subroutine to direct program control around it.

Use ON–GOSUB to branch to different subroutines based on the result of an expression.

Example:

```
Ok
10 GOSUB 40
20 PRINT "BACK FROM SUBROUTINE"
30 END
40 PRINT "SUBROUTINE";
50 PRINT " IN";
60 PRINT " PROGRESS"
70 RETURN
RUN
SUBROUTINE IN PROGRESS
BACK FROM SUBROUTINE
Ok
```

This example shows how a subroutine works. The GOSUB in line 10 calls the subroutine in line 40. So the program branches to line 40 and starts executing statements there until it sees the RETURN statement in line 70. At that point the program goes back to the statement after the subroutine call; that is, it returns to line 20. The END statement in line 30 prevents the subroutine from being performed a second time.

GOTO
Statement

Purpose: Branches unconditionally out of the normal program sequence to a specified line number.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: GOTO *line*

Remarks:

line is the line number of a line in the program.

If *line* is the line number of an executable statement, that statement and those following are executed. If *line* refers to a nonexecutable statement (such as REM or DATA), the program continues at the first executable statement after *line*.

The GOTO statement can be used in direct mode to re-enter a program. This can be useful in debugging.

Use ON–GOTO to branch to different lines based on the result of an expression.

Example:

```
Ok
5  DATA 5,7,12
10 READ R
20 PRINT "R =";R,
30 A = 3.14*R^2
40 PRINT "AREA =";A
50 GOTO 5
RUN
R = 5           AREA = 78.5
R = 7           AREA = 153.86
R = 12          AREA = 452.16
Out of data in 10
Ok
```

The GOTO statement in line 50 puts the program into a continuous loop. The loop stops when the program runs out of data in the DATA statement. (Notice how branching to the DATA statement did not add values to the internal data table.)

HEX$ Function

Purpose: Returns a string which represents the hexadecimal value of the decimal argument.

Versions: Cassette *** Cartridge *** Compiler ***

Cassette	Cartridge	Compiler
***	***	***

Format: *v$* = HEX$(*n*)

Remarks:

n is a numeric expression in the range -32768 to 65535.

If *n* is negative, the two's complement form is used. That is, HEX$(-*n*) is the same as HEX$(65536-*n*).

See the OCT$ function for octal conversion.

Example: The following example uses the HEX$ function to figure the hexadecimal representation for the two decimal values which are entered.

```
Ok
10 INPUT X
20 A$ = HEX$(X)
30 PRINT X "DECIMAL IS " A$ " HEXADECIMAL"
RUN
? 32
 32 DECIMAL IS 20 HEXADECIMAL
Ok
RUN
? 1023
 1023 DECIMAL IS 3FF HEXADECIMAL
Ok
```

Purpose: Makes a decision about program flow based on the result of an expression.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: IF *expression* [,]THEN *clause* [ELSE *clause*]

IF *expression* [,]GOTO *line* [[,]ELSE *clause*]

Remarks:

expression may be any numeric expression.

clause may be a BASIC statement or a sequence of statements (separated by colons); or it may be simply the number of a line to branch to.

line is the line number of a line existing in the program.

If the *expression* is true (not zero), the THEN or GOTO clause is executed. THEN may be followed by either a line number for branching or one or more statements to be executed. GOTO is always followed by a line number.

If the result of *expression* is false (zero), the THEN or GOTO clause is ignored and the ELSE clause, if present, is executed. Execution then continues with the next numbered line containing an executable statement.

If you enter an IF–THEN statement in direct mode, and it directs control to a line number, then an

Undefined line number error results unless you previously entered a line with the specified line number.

Note: When using IF to test equality for a value that is the result of a single- or double-precision computation, remember that the internal representation of the value may not be exact. (This is because single- and double-precision values are stored in floating point binary format.) Therefore, the test should be against the *range* over which the accuracy of the value may vary. For example, to test a computed variable A against the value 1.0, use:

```
IF ABS (A-1.0)<1.0E-6 THEN ...
```

This test returns a true result if the value of A is 1.0 with a relative error of less than 1.0E-6.

Also note that IF–THEN–ELSE is just *one statement*. Once an IF statement occurs on a line, everything else on that line is part of the IF statement. Because IF–THEN–ELSE is all one statement, the ELSE clause cannot be a separate program line. For example:

```
10 IF A=B THEN X=4
20 ELSE P=Q
```

is invalid. Instead, it should be:

```
10 IF A=B THEN X=4 ELSE P=Q
```

Nesting of IF Statements: IF–THEN–ELSE statements may be nested. Nesting is limited only by the length of the line. For example,

```
IF X>Y THEN PRINT "GREATER" ELSE IF Y>X
   THEN PRINT "LESS THAN" ELSE PRINT "EQUAL"
```

is a valid statement. If the statement does not contain the same number of ELSE and THEN clauses, each ELSE is matched with the closest unmatched THEN. For example:

```
IF A=B THEN IF B=C THEN PRINT "A=C"
     ELSE PRINT "A<>C"
```

will not print "A<>C" when A<>B.

Example: This statement gets record I if I is not zero:

```
200 IF I THEN GET #1,I
```

In the next example, if I is between 10 and 20, DB is calculated, and execution branches to line 300. If I is not in this range, the message **OUT OF RANGE** is printed. Note the use of two statements in the THEN clause.

```
100 IF (I>10) AND (I<20) THEN
        DB=1982-I: GOTO 300
    ELSE PRINT "OUT OF RANGE"
```

This next statement causes printed output to go to either the screen or the printer, depending on the value of a variable (IOFLAG). If IOFLAG is false (zero), output goes to the printer; otherwise, output goes to the screen:

```
210 IF IOFLAG THEN PRINT A$ ELSE LPRINT A$
```

In line 20 of the following example everything after the THEN is part of the clause. This means that the NEXT is not executed unless N=1. When line 20 executes, N does not equal I so the IF evaluation is false. Therefore, the NEXT is not performed and the program falls through to line 30. The NEXT must be coded on a separate line if you want the program to loop until N=I.

```
10 N=15
20 FOR I=1 TO 20:IF N=I THEN 40:NEXT
30 PRINT "N <> I":END
40 PRINT "N = I"
RUN

N <> I
Ok
```

See “IF–THEN” in Appendix D, “Converting Programs to PCjr BASIC” for information on how the IF statement in PCjr BASIC differs from the IF statement in other BASICs.

INKEY$ Variable

Purpose: Reads a character from the keyboard.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: *v$* = INKEY$

Remarks: INKEY$ only reads a single character, even if there are several characters waiting in the keyboard buffer. The returned value is a zero-, one-, or two-character string.

- A null string (length zero) indicates that no character is pending at the keyboard.

- A one-character string contains the actual character read from the keyboard.

- A two-character string indicates a special extended code. The first character will be hex 00. For a complete list of these codes, see Appendix G, "ASCII Character Codes."

You must assign the result of INKEY$ to a string variable before using the character with any BASIC statement or function.

While INKEY$ is being used, no characters are displayed on the screen and all characters are passed through to the program except for:

- Fn followed by Break, which stops the program

- Fn followed by Pause, which sends the system into a pause state

- Alt-Ctrl-Del, which does a System Reset

- Fn followed by PrtSc, which prints the screen

If you press Enter in response to INKEY$, the carriage return character passes through to the program.

Example: The following section of a program stops the program until any key on the keyboard is pressed:

```
110 PRINT "Press any key to continue"
120 A$=INKEY$: IF A$="" THEN 120
```

The next example shows program lines that could be used to test a two-character code being returned:

```
210 KB$=INKEY$
220 IF LEN(KB$)=2 THEN KB$=RIGHT$(KB$,1)
```

INP Function

Purpose: Returns the byte read from port *n*.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: *v* = INP(*n*)

Remarks: *n* must be in the range 0 to 65535.

INP is the complementary function to the OUT statement (see "OUT Statement" in this chapter).

INP performs the same function as the IN instruction in assembly language. Refer to the PCjr *Technical Reference* manual for a description of valid port numbers (I/O addresses).

Example:

```
100 A=INP(255)
```

This instruction reads a byte from port 255 and assigns it to the variable A.

INPUT Statement

Purpose: Receives input from the keyboard during program execution.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: INPUT[;]["*prompt*";] *variable*[,*variable*]...

Remarks:

"*prompt*" is a string constant which will be used to prompt you for input.

variable is the name of the numeric or string variable or array element which will receive the input.

When the program sees an INPUT statement, it pauses and displays a question mark on the screen to indicate that it is waiting for data. If a "*prompt*" is included, the string is displayed before the question mark. You may then enter the required data from the keyboard.

You may use a comma instead of a semicolon after the prompt string to stop the question mark from printing. For example, the statement INPUT "ENTER BIRTHDATE: ",B$ prints the prompt without the question mark.

The data that you enter is assigned to the *variable*(s) given in the variable list. The data items you supply must be separated by commas, and the number of data items must be the same as the number of variables in the list.

The type of each data item that you enter must agree with the type specified by the variable name. (Strings entered in response to an INPUT statement need not be surrounded by quotation marks.)

If you respond to INPUT with too many or too few items, or with the wrong type of value (letters instead of numbers, etc.), BASIC displays the message **?Redo from start**. If a single variable is requested, you may simply press Enter to indicate the default values of 0 for numeric input or null for string input. However, if more than one variable is requested, pressing Enter causes the **?Redo from start** message to be printed because too few items were entered. BASIC does not assign any of the input values to variables until you give an acceptable response.

In Cartridge BASIC, if INPUT is immediately followed by a semicolon, then pressing Enter to input data does not produce a carriage return/line feed sequence on the screen. This means that the cursor remains on the same line as your response.

Example:

```
Ok
10 INPUT X
20 PRINT X "SQUARED IS" X^2
30 END
RUN
?
```

In this example, the question mark displayed by the computer is a prompt to tell you it wants you to enter something. Suppose you enter a 5. The program continues:

```
.
.
.
?  5
5 SQUARED IS 25
Ok

Ok
10 PI=3.14
20 INPUT "WHAT IS THE RADIUS";R
30 A=PI*R^2
40 PRINT "THE AREA OF THE CIRCLE IS";A
50 END
RUN
WHAT IS THE RADIUS?
```

For this second example, a prompt was included in line 20, so this time the computer prompts with "WHAT IS THE RADIUS? ". Suppose you respond with 7.4. The program continues:

```
.
.
.
WHAT IS THE RADIUS?  7.4
THE AREA OF THE CIRCLE IS 171.9464
Ok
```

INPUT # Statement

Purpose: Reads data items from a sequential device or file and assigns them to program variables.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: INPUT #*filenum, variable* [,*variable*]...

Remarks:

filenum is the number used when the file was opened for input.

variable is the name of a variable that will have an item in the file assigned to it. It may be a string or numeric variable, or an array element.

The sequential file may reside on diskette or on cassette; it may be a sequential data stream from a communications adapter; or it may be the keyboard (KYBD:).

The type of data in the file must match the type specified by the variable name. Unlike INPUT, no question mark is displayed with INPUT #.

The data items in the file should appear just as they would if the data were being typed in response to an INPUT statement. With numeric values, leading spaces, carriage returns, and line feeds are ignored. The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of the number. The number ends with a space, carriage return, line feed, or comma.

If BASIC is scanning the data for a string item, leading spaces, carriage returns, and line feeds are also ignored. The first character encountered that is not a space, carriage return, or line feed is assumed to be the start of the string item. If this first character is a quotation mark ("), the string item will consist of all characters read between the first quotation mark and the second. Thus, a quoted string may not contain a quotation mark as a character. If the first character of the string is not a quotation mark, the string is an unquoted string; it will end when a comma, carriage return, or line feed, or after 255 characters have been read. If end of file is reached when a numeric or string item is being input, the item is canceled.

INPUT # can also be used with a random file.

Example: See Appendix B.

Purpose: Returns a string of *n* characters, read from the keyboard or from file number *filenum*.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: *v$* = INPUT$(*n*[,[#]*filenum*])

Remarks:

n is the number of characters to be read from the file.

filenum is the file number used on the OPEN statement. If *filenum* is omitted, the keyboard is read.

If the keyboard is used for input, no characters will be displayed on the screen. All characters (including control characters) are passed through except Fn/Break, which is used to interrupt the execution of the INPUT$ function. When responding to INPUT$ from the keyboard, it is not necessary to press Enter.

The INPUT$ function allows you to read characters from the keyboard which are significant to the BASIC program editor, such as Backspace (ASCII code 8). If you want to read these special characters, you should use INPUT$ or INKEY$ (*not* INPUT or LINE INPUT).

For communications files, the INPUT$ function is preferred over the INPUT # and LINE INPUT # statements, since all ASCII characters may be significant in communications. Refer to Appendix F, "Communications."

INPUT$
Function

Example: The following program lists the contents of a sequential file in hexadecimal.

```
10 OPEN "DATA" FOR INPUT AS #1
20 IF EOF(1) THEN 50
30 PRINT HEX$(ASC(INPUT$(1,#1)));
40 GOTO 20
50 PRINT
60 END
```

The next example reads a single character from the keyboard in response to a question.

```
100 PRINT "TYPE P TO PROCEED OR S TO STOP"
110 X$=INPUT$(1)
120 IF X$="P" THEN 500
130 IF X$="S" THEN 700 ELSE 100
```

INSTR Function

Purpose: Searches for the first occurrence of string *y$* in *x$* and returns the position at which the match is found. The optional offset *n* sets the position for starting the search in *x$*

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: *v* = INSTR([*n*,]*x$*,*y$*)

Remarks:

n is a numeric expression from 1 to 255.

x$*, *y$ may be string variables, string expressions or string constants.

If *n*>LEN(*x$*), or if *x$* is null, or if *y$* cannot be found, INSTR returns 0. If *y$* is null, INSTR returns *n* (or 1 if *n* is not specified).

If *n* is out of range, an **Illegal function call** occurs.

Example: This example searches for the string "B" within the string "ABCDEB". When the string is searched from the beginning, "B" is found at position 2; when the search starts at position 4, "B" is found at position 6.

```
Ok
10 A$ = "ABCDEB": B$="B"
20 PRINT INSTR(A$,B$);INSTR(4,A$,B$)
RUN
 2  6
Ok
```

STATEMENTS

INT
Function

Purpose: Returns the largest integer that is less than or equal to *x*.

Versions: Cassette *** Cartridge *** Compiler ***

Format: *v* = INT(*x*)

Remarks: *x* is any numeric expression.

This is called the "floor" function in some other programming languages.

See the FIX and CINT functions, which also return integer values.

Example:

```
Ok
PRINT INT(45.67)
 45
Ok
PRINT INT(-2.89)
-3
Ok
```

This example shows how INT truncates positive integers, but rounds negative numbers upward (in a negative direction).

Purpose: Sets or displays the soft keys.

Versions:

Cassette	Cartridge	Compiler
***	***	(**)

Format: KEY ON

KEY OFF

KEY LIST

KEY *n, x$*

Remarks:

n is the function key number in the range 1 to 10.

x$ is a string expression which will be assigned to the key. (Remember to enclose string *constants* in quotation marks.)

The KEY statement allows function keys to be designated *soft keys*. That is, you can set each function key to automatically type any sequence of characters. A string of up to 15 characters may be assigned to any one or all the ten function keys. When the key is pressed, the string will be input to BASIC.

KEY
Statement

Initially, the soft keys are assigned the following values:

F1	LIST	F6	,"LPT1:"←
F2	RUN←	F7	TRON←
F3	LOAD"	F8	TROFF←
F4	SAVE"	F9	KEY
F5	CONT←	F10	SCREEN 0,0,0←

The arrow (←) indicates Enter.

KEY ON causes the soft key values to be displayed on the 25th line. When the width is 20, two of the ten soft keys are displayed. When the width is 40, five of the ten soft keys are displayed. When the width is 80, all ten are displayed. In any width, only the first six characters of each value are displayed. ON is the default state for the soft key display.

KEY OFF erases the soft key display from the 25th line, making that line available for program use. It does not disable the function keys.

After turning off the soft key display with KEY OFF, you can use LOCATE 25,1 followed by PRINT to display anything you want on the bottom line of the screen. Information on line 25 is not scrolled, as are lines 1 through 24.

KEY LIST lists all ten soft key values on the screen. All 15 characters of each value are displayed.

KEY *n, x$* assigns the value of *x$* to the function key specified (1 to 10). *x$* may be 1 to 15 characters in length. If it is longer than 15 characters, only the first 15 characters are assigned.

Assigning a null string (string of length zero) to a soft key disables the function key as a soft key.

KEY
Statement

If the value entered for *n* is not in the range 1 to 10, an **Illegal function call** error occurs. The previous key string assignment is retained.

When a soft key is pressed, the INKEY$ function returns one character of the soft key string each time it is called. If the soft key is disabled, INKEY$ returns a two character string. The first character is binary zero, the second is the key scan code, as listed in Appendix G, "ASCII Character Codes."

In Cartridge BASIC, there are six additional definable key traps. This lets you trap any Ctrl, Shift, or super-shift key. These additional keys are defined by the statement:

```
KEY n,CHR$(shift)+CHR$(scan code)
```

n is a numeric expression in the range 15 to 20.

shift is a numeric value that corresponds to the hex value for the latched keys. The hex values for each key are:

Caps Lock	&H40
Alt	&H08
Ctrl	&H04
Shift	&H01, &H02, &H03

Note that key trapping assumes that the left and right Shift keys are the same, so you can use a value of &H01, &H02, or &H03 (the sum of hex 01 and hex 02) to denote a Shift key.

You can also add multiple shift states together, such as the Ctrl and Alt keys added together. Shift state values *must* be in hex.

scan code is a number in the range 1 to 83 that identifies the key to be trapped. See Appendix K, "Keyboard Diagram and Scan Codes," for a complete table of scan codes and their associated key positions.

When you trap keys, they are processed in the following order:

1. Fn followed by Echo, which activates the line printer, is processed first. To trap this you will need to trap on Ctrl followed by scan code 55. Even if Fn-Echo is defined as a trappable key combination, it can still be pressed to echo display output to the printed.

2. Next, the function keys Fn plus F1 to F10, Cursor Up, Cursor Down, Cursor Right, and Cursor Left (1-14) are processed. Setting scan codes 59 to 68, 72, 75, 77, or 80 as key traps has no effect, because they are considered to be predefined.

3. Last, the keys you define for 15 to 20 are processed.

Notes:

1. Trapped keys do not go into the keyboard buffer.

2. You cannot trap Fn by itself, Fn/Pause, Fn/PrtSc, or some three key combination.

3. To trap Fn/Sc Lock you will need to trap on scan code 70.

4. To trap Fn/Break you will need to trap on Ctrl followed by scan code 71.

5. Be careful when you trap Fn-Break and Ctrl-Alt-Del, because unless you have a test in your trap routine, you will have to turn the power off to stop your program.

See the following section, "KEY(n) Statement," to see how to enable and disable function key trapping in Cartridge BASIC.

Examples:

```
10 KEY ON
```

displays the soft keys on the 25th line.

```
200 KEY OFF
```

erases soft key display. The soft keys are still active, but not displayed.

```
10 KEY 1,"FILES"+CHR$(13)
```

assigns the string "FILES"+Enter to soft key 1. This is a way to assign a commonly used command to a function key.

```
20 KEY 1,""
```

disables function key 1 as a soft key.

```
100 KEY 15, CHR$(&H40)+CHR$(25)
100 ON KEY(15) GOSUB 1000
120 KEY(15) ON
```

sets up a key trap for capital P. Note that all three KEY statements—KEY, KEY(n), and ON KEY—are used with key trapping.

```
200 KEY 20, CHR$(&H04+&H03)+CHR$(30)
210 ON KEY(20) GOSUB 2000
220 KEY(20) ON
```

sets up a key trap for Ctrl-Shift-A. Notice that the hex values for Ctrl (&H04) and Shift (&H03) are added together to get the shift state.

KEY(n) Statement

Purpose: Activates and deactivates trapping of the specified key in a BASIC program. See "ON KEY(n) Statement" in this chapter.

Versions:

Cassette	Cartridge	Compiler
	***	(**)

Format: KEY(*n*) ON

KEY(*n*) OFF

KEY(*n*) STOP

Remarks:

n is a numeric expression in the range 1 to 20, and indicates the key to be trapped:

1-10	function keys Fn plus F1 to F10
11	Cursor Up
12	Cursor Left
13	Cursor Right
14	Cursor Down
15-20	keys defined by the form: KEY *n*,CHR$(*shift*)+CHR$(*scan code*)

STATEMENTS

KEY(*n*) ON must be executed to activate trapping of function key or cursor control key activity. After KEY(*n*) ON, if a non-zero line number was specified in the ON KEY(*n*) statement then every time BASIC starts a new statement it will check to see if the specified key was pressed. If so it will perform a GOSUB to the line number specified in the ON KEY(*n*) statement. A KEY(*n*) statement cannot precede an ON KEY(*n*) statement.

KEY(n) Statement

If KEY(*n*) is OFF, no trapping takes place and even if the key is pressed, the event is not remembered.

Once a KEY(*n*) STOP statement has been executed, no trapping will take place. However, if you press the specified key your action is remembered so that an immediate trap takes place when KEY(*n*) ON is executed.

KEY(*n*) ON has no effect on whether the soft key values are displayed at the bottom of the screen.

If you use a KEY(n) statement in Cassette you will get a **Syntax error**. Refer to the previous section, "KEY Statement," for an explanation of the KEY statement.

Purpose: Deletes a file from a diskette. The KILL command in BASIC is similar to the ERASE command in DOS.

Note: This command requires the use of DOS 2.10. If DOS 2.10 is not present, an **Illegal function call** error will occur.

Versions:

Cassette	Cartridge	Compiler
	***	***

Format: KILL *filespec*

Remarks:

filespec is a valid file specification as explained under "Naming Files" in Chapter 3. The device name must be a diskette drive. If the device name is omitted, the DOS default drive is used.

In Cartridge BASIC, *filespec* can also be a path, as explained under "Naming Files" in Chapter 3.

KILL can be used for all types of diskette files. The name must include the extension, if one exists. For example, you may save a BASIC program using the command

```
SAVE "TEST"
```

BASIC supplies the extension **.BAS** for the SAVE command, but not for the KILL command. If you want to delete that program file later, you must say KILL "TEST.BAS", not KILL "TEST".

KILL Command

If a KILL statement is given for a file that is currently open, a **File already open** error occurs.

Example: To delete the file named "DATA1" on drive A, you might use:

```
200 KILL "A:DATA1"
```

To delete the file "PROG.BAS" in the LEVEL2 sub-directory, you might use:

```
KILL "LEVEL1\LEVEL2\PROG.BAS"
```

Note that KILL can only be used to delete files. The RMDIR command must be used to remove directories.

LEFT$ Function

Purpose: Returns the leftmost *n* characters of *x$*.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: *v$* = LEFT$(*x$*,*n*)

Remarks:

x$ is any string expression.

n is a numeric expression which must be in the range 0 to 255.It specifies the number of characters which are to be in the result.

If *n* is greater than LEN(*x$*), the entire string (*x$*) is returned. If *n*=0, the null string (length zero) is returned.

Also see the MID$ and RIGHT$ functions.

Example:

```
Ok
10 A$ = "BASIC PROGRAM"
20 B$ = LEFT$(A$,5)
30 PRINT B$
RUN
BASIC
Ok
```

In this example, the LEFT$ function is used to extract the first five characters from the string "BASIC PROGRAM."

LEN Function

Purpose: Returns the number of characters in *x$*.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: *v* = LEN(*x$*)

Remarks: *x$* is any string expression.

Unprintable characters and blanks are included in the count of the number of characters.

Example:

```
10 X$ = "BOCA RATON, FL"
20 PRINT LEN(X$)
RUN
 14
Ok
```

There are 14 characters in the string "BOCA RATON, FL," because the comma and the blank are counted.

Purpose: Assigns the value of an expression to a variable.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: [LET] *variable=expression*

Remarks:

variable is the name of the variable or array element which is to receive a value. It may be a string or numeric variable or array element.

expression is the expression whose value will be assigned to *variable*. The type of the expression (string or numeric) must match the type of the variable, or a **Type mismatch** error will occur.

The use of the word LET is optional when assigning an expression to a variable name. The equals sign can be by itself to produce the same results.

Example:

```
110 LET DORI=12
120 LET E=DORI+2
130 LET FDANCE$="HORA"
```

This example assigns the value 12 to the variable DORI. It then assigns the value 14, which is the value of the expression DORI+2, to the variable E. The string "HORA" is assigned to the variable FDANCE$.

The same statements could have also been written:

LET
Statement

```
110 DORI= 12
120 E =DORI+2
130 FDANCE$ = "HORA"
```

Purpose: Draws a line or a box on the screen.

Versions:

Cassette	Cartridge	Compiler
***	***	(**)

Graphics mode only.

Format: LINE [(*x1,y1*)] -(*x2,y2*) [,[*attribute*] [,B[F]] [,*style*]]

Remarks:

(x1,y1), (x2,y2)
are coordinates in either absolute or relative form. (See "Specifying Coordinates"under "Graphics Modes" in Chapter 3.)

attribute is an integer or integer expression in the range of 0 to 15. In low resolution, there are 16 attributes available (0 to 15). In medium resolution, there are 4 (0 to 3) or 16 (0 to 15) attributes available, depending on the current screen mode. In high resolution, there 2 (0 to 1) or 4 (0 to 3) attributes available, depending on the current screen mode. The default attribute is always the maximum attribute for the current screen mode. 0 is the background attribute. For more information see "Graphics Modes" in Chapter 3.

style is a 16-bit integer mask used to put points on the screen. The *style* option is used for normal lines and boxes, but cannot be used with filled boxes (BF).

Using *style* with BF results in a **Syntax error**. This technique, called line styling, is for use in Cartridge BASIC only.

The simplest form of LINE is:

```
LINE -(X2,Y2)
```

This will draw a line from the last point referenced to the point (X2,Y2) in the foreground attribute.

We can include a starting point also:

```
LINE (0,0)-(319,199) 'diagonal down screen
LINE (0,100)-(319,100) 'bar across screen
```

We can indicate the color to draw the line in:

```
LINE (10,10)-(20,20),2 'draw in color 2
```

```
1 'draw random lines in random colors
10 SCREEN 1,0,0,0: CLS
20 LINE -(RND*319,RND*199),RND*4
30 GOTO 20
```

```
1 'alternating pattern - line on, line off
10 SCREEN 1,0,0,0: CLS
20 FOR X=0 TO 319
30 LINE (X,0)-(X,199),X AND 1
40 NEXT
```

The next argument to LINE is **B** (box), or **BF** (filled box). We can leave out *color* and include the argument:

```
LINE (0,0)-(100,100),,B 'box in foreground
```

or we may include the color:

```
LINE (0,0)-(100,100),2,BF 'filled box color 2
```

The **B** tells BASIC to draw a rectangle with the points (*x1,y1*) and (*x2,y2*) as opposite corners. This avoids having to give the four LINE commands:

```
LINE (X1,Y1)-(X2,Y1)
LINE (X1,Y1)-(X1,Y2)
LINE (X2,Y1)-(X2,Y2)
LINE (X1,Y2)-(X2,Y2)
```

which perform the equivalent function.

The **BF** means draw the same rectangle as **B**, but also fill in the interior points with the selected color.

The last argument to line is *style*. LINE uses the current circulating bit in *style* to plot (or store) points on the screen. If the bit is 0 (zero), no point is plotted. If the bit is 1 (one), a point is plotted. After each point, the next bit position in *style* is selected. When the last bit position in *style* is selected, LINE "wraps around" and begins with the first bit position again.

Note that a 0 (zero) bit indicates *no store* and does not erase the existing point on the screen. You may want to draw a background line before a styled line to force a known background.

The *style* option can be used to draw a dotted line across the screen by plotting (storing) every other point. Because *style* is 16 bits wide, the pattern for a dotted line looks like this:

```
1 0 1 0 1 0 1 0 1 0 1 0 1 0 1 0
```

This is equal to AAAA in hexadecimal notation. For help in choosing the correct hexadecimal values, see Appendix H, "Hexadecimal Conversion Tables."

Examples: To draw a dotted line:

```
10 SCREEN 1,0
20 LINE (0,0)-(319,199),,,&HAAAA
```

To draw a cyan box with dashes:

```
10 SCREEN 1,0
20 LINE (0,0)-(100,100),1,B,&HCCCC
```

In Cartridge BASIC, out-of-range coordinates are not visible on the viewing surface. This is called *line clipping* because the image that is outside of the coordinate range is "clipped" at the boundaries of the viewing surface.

The last point referenced after a LINE statement is point (*x2,y2*). If you use the relative form for the second coordinate, it is relative to the first coordinate. For example,

```
LINE (100,100)-STEP (10,-20)
```

will draw a line from (100,100) to (110,80).

This example will draw random filled boxes in random colors.

```
10 CLS
20 SCREEN 1,0: COLOR 0,0
30 LINE -(RND*319,RND*199),RND*2+1,BF
40 GOTO 30 'boxes will overlap
```

LINE INPUT Statement

Purpose: Reads an entire line (up to 254 characters) from the keyboard into a string variable, ignoring delimiters.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: LINE INPUT[;][*"prompt"*;] *stringvar*

Remarks:

"prompt" is a string constant that is displayed on the screen before input is accepted. A question mark is not printed unless it is part of the prompt string.

stringvar is the name of the string variable or array element to which the line will be assigned. All input from the end of the prompt to the Enter is assigned to *stringvar*. Trailing blanks are ignored.

In Cartridge BASIC, if LINE INPUT is immediately followed by a semicolon, then pressing Enter to end the input line does not produce a carriage return/line feed sequence on the screen. That is, the cursor remains on the same line as your response.

You can exit LINE INPUT by pressing the Fn key followed by Break key. BASIC returns to command level and displays **Ok**. You may then enter CONT to resume execution at the LINE INPUT.

Example: See example in the next section, "LINE INPUT # Statement."

LINE INPUT # Statement

Purpose: Reads an entire line (up to 254 characters), ignoring delimiters, from a sequential file into a string variable.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: LINE INPUT #*filenum, stringvar*

Remarks:

filenum is the number under which the file was opened.

stringvar is the name of a string variable or array element to which the line will be assigned.

LINE INPUT # reads all characters in the sequential file up to a carriage return. It then skips over the carriage return/line feed sequence, and the next LINE INPUT # reads all characters up to the next carriage return. (If a line feed/carriage return sequence is encountered, it is preserved. That is, the line feed/carriage return characters are returned as part of the string.)

LINE INPUT # is especially useful if each line of a file has been broken into fields, or if a BASIC program saved in ASCII mode is being read as data by another program.

LINE INPUT # can also be used for random files. See Appendix B, "BASIC Diskette Input and Output."

Example: The following example uses LINE INPUT to get information from the keyboard, where the information is likely to have commas or other delimiters in it. Then the information is written to a sequential file, and read back out from the file using LINE INPUT #.

```
Ok
10 OPEN "LIST" FOR OUTPUT AS #1
20 LINE INPUT "Address? ";C$
30 PRINT #1, C$
40 CLOSE 1
50 OPEN "LIST" FOR INPUT AS #1
60 LINE INPUT #1, C$
70 PRINT C$
80 CLOSE 1
RUN

Address?
```

Suppose you respond with DELRAY BEACH, FL 33445. The program continues:

```
.
.
.
Address? DELRAY BEACH, FL    33445

DELRAY BEACH, FL    33445
Ok
```

LIST
Command

Purpose: Lists the program currently in memory on the screen or other specified device.

Versions:

Cassette	Cartridge	Compiler
***	***	

Format: LIST [*line1*] [-[*line2*]] [, *filespec*]

Remarks:

line1, line2 are valid line numbers in the range 0 to 65529. *line1* is the first line to be listed. *line2* is the last line to be listed. A period (.) may be used for either line number to indicate the current line.

filespec is a string expression for the file specification as outlined under "Naming Files" in Chapter 3. If *filespec* is omitted, the specified lines are listed on the screen.

Any listing to either the screen or the printer may be interrupted by pressing the Fn key followed by the Break key.

If the line range is omitted, the entire program is listed.

When the dash (-) is used in a line range, three options are available:

- If only *line1* is given, that line and all higher numbered lines are listed.
- If only *line2* is given, all lines from the beginning of the program through *line2* are listed.
- If both line numbers are specified, all lines from *line1* through *line2*, inclusive, are listed.

When you list to a file on cassette or diskette, the specified part of the program is saved in ASCII format. This file may later be used with MERGE.

BASIC always returns to the command level after a LIST is executed.

Example: To list the entire program on the screen:

```
LIST
```

To list line 35 on the screen:

```
LIST 35,"SCRN:"
```

To list lines 10 through 20 on the printer:

```
LIST 10-20, "LPT1:"
```

To list from the first line through line 200 to a file named "BOB" on cassette:.

```
LIST -200,"CAS1:BOB"
```

LLIST Command

Purpose: Lists all or part of the program currently in memory on the printer (LPT1:).

Versions:

Cassette	Cartridge	Compiler
***	***	

Format: LLIST [*line1*][- [*line2*]]

Remarks: The line number ranges for LLIST work the same as for LIST.

BASIC always returns to command level after an LLIST is executed.

Example: To print a listing of the entire program:

```
LLIST
```

To print line 35:

```
LLIST 35
```

To list lines 10 through 20 on the printer:

```
LLIST 10-20
```

To print all lines from line 100 through the end of the program:

```
LLIST 100-
```

To print the first line through line 200:

```
LLIST -200
```

Purpose: Loads a program from the specified device into memory, and optionally runs it.

Versions:

Cassette	Cartridge	Compiler
***	***	

Format: LOAD *filespec*[,R]

Remarks:

filespec is a string expression for the file specification. It must conform to the rules outlined under "Naming Files" in Chapter 3, otherwise an error occurs and the load is canceled.

LOAD closes all open files and deletes all variables and program lines currently residing in memory before it loads the specified program. If the **R** option is omitted, BASIC returns to direct mode after the program is loaded.

However, if the **R** option is used with LOAD, the program is run after it is loaded. In this case all open data files are kept open. Thus, LOAD with the **R** option may be used to chain several programs (or segments of the same program). Information may be passed between the programs using data files.

LOAD *filespec*,R is equivalent to RUN *filespec*.

If you are using Cassette BASIC and the device name is omitted, CAS1: is assumed. CAS1: is the only allowable device for LOAD in Cassette BASIC and for Cartridge BASIC without DOS.

LOAD Command

If you are using Cartridge BASIC and DOS is present, the DOS default diskette drive is used if the device is omitted.

The extension **.BAS** is added to the filename if no extension is supplied and the filename is eight characters or less.

Notes when using CAS1:

1. If the LOAD statement is entered in direct mode, the file names on the tape will be displayed on the screen followed by a period (.) and a single letter indicating the type of file. This is followed by the message **Skipped** for the files not matching the named file, and **Found** when the named file is found. Types of files and their corresponding letter are:

   .B for BASIC programs in internal format (created with SAVE command)
   .P for protected BASIC programs in internal format (created with SAVE ,P command)
   .A for BASIC programs in ASCII format (created with SAVE ,A command)
   .M for memory image files (created with BSAVE command)
   .D for data files (created by OPEN followed by output statements)

   To see what files are on a cassette tape, rewind the tape and enter some name that is known not to be on the tape. For example, **LOAD "CAS1:NOWHERE"**. All file names will then be displayed.

If the LOAD command is executed in a BASIC program, the file names skipped and found are not displayed on the screen.

2. Note that Fn and Break keys may be typed at any time during LOAD. Between files or after a time-out period, BASIC will exit the search and return to command level. Previous memory contents remain unchanged.

3. If CAS1: is specified as the device and the filename is omitted, the next program file on the tape is loaded.

Example:

```
LOAD "MENU"
```

Loads the program named MENU, but does not run it.

```
LOAD "INVENT",R
```

Loads and runs the program INVENT.

```
RUN "INVENT"
```

Same as LOAD "INVENT",R.

```
LOAD "A:REPORT.BAS"
```

Loads the file REPORT.BAS from diskette drive A. Note that the **.BAS** did not have to be specified.

```
LOAD "CAS1:"
```

Loads the next program on the tape.

LOC
Function

Purpose: Returns the current position in the file.

Note: This function requires the use of DOS 2.10.

Versions:

Cassette	Cartridge	Compiler
	***	***

Format: *v* = LOC(*filenum*)

Remarks:

filenum is the file number used when the file was opened.

With random files, LOC returns the record number of the last record read or written to a random file.

With sequential files, LOC returns the number of records read from or written to the file since it was opened. (A record is a 128 byte block of data.) When a file is opened for sequential input, BASIC reads the first sector of the file, so LOC will return a 1 even before any input from the file.

For a communications file, LOC returns the number of characters in the input buffer waiting to be read. The default size for the input buffer is 256 characters, but you can change this with the **/C:** option on the BASIC command. If there are more than 255 characters in the buffer, LOC returns 255. Since a string is limited to 255 characters, this practical limit alleviates the need for you to test for string size before reading data into it. If fewer than 255 characters remain in the buffer, then LOC returns the actual count.

Example:

```
200 IF LOC(1)>50 THEN STOP
```

This first example stops the program if we've gone past the 50th record in the file.

```
300 PUT #1,LOC(1)
```

The second example could be used to re-write the record that was just read.

LOCATE Statement

Purpose: Places the cursor on the active screen. Optional parameters turn the blinking cursor on and off and define the size of the blinking cursor.

Versions: Cassette *** Cartridge *** Compiler ***

Format: LOCATE [*row*][,[*col*][,[*cursor*][,[*start*] [,*stop*]]]]

Remarks:

row is a numeric expression in the range 1 to 25. It indicates the screen line number where you want to place the cursor.

col is a numeric expression in the range 1 to 20, 1 to 40 or 1 to 80, depending upon screen width. It indicates the screen column number where you want to place the cursor.

cursor is a value indicating whether the cursor is visible or not. A 0 (zero) indicates off, 1 (one) indicates on.

start is the cursor starting scan line. It must be a numeric expression in the range 0 to 31.

stop is the cursor stop scan line. It also must be a numeric expression in the range 0 to 31.

cursor, start and *stop* do not apply to graphics mode.

start and *stop* allow you to make the cursor any size you want. You indicate the starting and ending scan lines. The scan lines are numbered from 0 at the top

of the character position. The bottom scan line is 7. If *start* is given and *stop* is omitted, *stop* assumes the value of *start*. If *start* is greater than *stop*, you will get a two-part cursor. The cursor "wraps" from the bottom line back to the top.

After a LOCATE statement, I/O statements to the screen begin placing characters at the specified location.

When a program is running, the cursor is normally off. You can use LOCATE ,,1 to turn it back on.

Normally, BASIC will not print to line 25. However, you can turn off the soft key display using KEY OFF, then use LOCATE 25,1: PRINT... to put things on line 25.

Any parameter may be omitted. Omitted parameters assume the current value.

Any values entered outside the ranges indicated will result in an **Illegal function call** error. Previous values are kept.

Examples:

```
10 LOCATE 1,1
```

Moves the cursor to the home position in the upper left-hand corner of the screen.

```
20 LOCATE ,,1
```

Makes the blinking cursor visible; its position remains unchanged.

```
30 LOCATE ,,,7
```

Position and cursor visibility remain unchanged. Sets the cursor to display at the bottom of the character. (starting and ending on scan line 7).

```
40 LOCATE 5,1,1,0,7
```

Moves the cursor to line 5, column 1. Makes the cursor visible, covering the entire character cell starting at scan line 0 and ending on scan line 7.

Purpose: Returns the number of bytes allocated to the file (length of the file).

Versions:

Cassette	Cartridge	Compiler
	***	***

Format: *v* = LOF(*filenum*)

Remarks:

filenum is the file number used when the file was opened.

For diskette files created by BASIC 1.10, LOF will return a multiple of 128. For example, if the actual data in the file is 257 bytes, the number 384 will be returned.

For diskette files created outside BASIC (for example, by using EDLIN) and for files created by Cartridge BASIC, LOF returns the actual number of bytes allocated to the file.

For communications, LOF returns the amount of free space in the input buffer. That is, *size*-LOC(*filenum*), where *size* is the size of the communications buffer, which defaults to 256 but may be changed with the **/C:** option on the BASIC command. Use of LOF may be used to detect when the input buffer is getting full. In practicality, LOC is adequate for this purpose, as demonstrated in the example in Appendix F, "Communications."

Example: These statements will get the last record of the file named BIG, assuming BIG was created with a record length of 128 bytes:

LOF
Function

```
10 OPEN "BIG" AS #1
20 GET #1,LOF(1)/128
```

Purpose: Returns the natural logarithm of *x*.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: *v* = LOG(*x*)

Remarks:

x must be a numeric expression which is greater than zero.

The natural logarithm is the logarithm to the base *e*.

LOG(*x*) can be calculated in single precision in Cassette BASIC and in either single or double precision in Cartridge BASIC.

Example: The first example calculates the logarithm of the expression 45/7:

```
Ok
PRINT LOG(45/7)
 1.860752
Ok
```

The second example calculates the logarithm of *e* and of e^2:

```
Ok
E= 2.718282
Ok
? LOG(E)
 1
Ok
? LOG(E*E)
 2
Ok
```

Purpose: Returns the current position of the print head within the printer buffer for LPT1:.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: *v* = LPOS(*n*)

Remarks:

n is a numeric expression which is a dummy argument in Cassette BASIC. In Cartridge BASIC, *n* indicates which printer is being tested, as follows:

0 or 1 LPT1:

The LPOS function does not necessarily give the physical position of the print head on the printer.

Example: In this example, if the line length is more than 60 characters long we send a carriage return character to the printer so it will skip to the next line.

```
100 IF LPOS(0)>60 THEN LPRINT CHR$(13)
```

LPRINT and LPRINT USING Statements

Purpose: Prints data on the printer (LPT1:).

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: LPRINT [*list of expressions*] [;]

LPRINT USING *v$; list of expressions* [;]

Remarks:

list of expressions
is a list of the numeric and/or string expressions that are to be printed. The expressions must be separated by commas or semicolons.

v$ is a string constant or variable which identifies the format to be used for printing. This is explained in detail under "PRINT USING Statement."

These statements function like PRINT and PRINT USING, except output goes to the printer. See "PRINT Statement" and "PRINT USING Statement."

LPRINT assumes an 80-character wide printer. That is, BASIC automatically inserts a carriage return/line feed after printing 80 characters. This will result in two lines being skipped when you print exactly 80 characters, unless you end the statement with a semicolon. You may change the width value with a WIDTH "LPT1:" statement.

If you do a form feed (LPRINT CHR$(12);) followed by another LPRINT and the printer takes

more than 20 seconds to do the form feed, you may get a **Device Timeout** error on the second LPRINT. To avoid this problem, do the following:

```
1 ON ERROR GOTO 65000
.
.
.
65000 IF ERR = 24 THEN RESUME '24=timeout
```

You might want to test ERL to make sure the timeout was caused by an LPRINT statement.

Example: This is an example of sending special control characters to the IBM 80 CPS Matrix Printer using LPRINT and CHR$. The printer control characters are listed in the the PC jr *Technical Reference* manual.

```
10 LPRINT CHR$(14);"      Title Line"
20 FOR I=2 TO 4
30 LPRINT "Report line";I
40 NEXT I
50 LPRINT CHR$(15);"Condensed print; 132 char/line"
60 LPRINT CHR$(18);"Return to normal"
70 LPRINT CHR$(27);"E";
80 LPRINT "This is emphasized print"
90 LPRINT CHR$(27);"F"
100 LPRINT "Back to normal again"
```

The output produced by this program looks like this:

4041

LSET and RSET Statements

Purpose: Moves data into a random file buffer (in preparation for a PUT (file) statement).

Versions:

Cassette	Cartridge	Compiler
	***	***

Format: LSET *stringvar* = *x$*

RSET *stringvar* = *x$*

Remarks:

stringvar is the name of a variable that was defined in a FIELD statement.

x$ is a string expression for the information to be placed into the field identified by *stringvar*.

If *x$* requires fewer bytes than were specified for *stringvar* in the FIELD statement, LSET left-justifies the string in the field, and RSET right-justifies the string. (Spaces are used to pad the extra positions.) If *x$* is longer than *stringvar*, characters are dropped from the right.

Numeric values must be converted to strings before they are LSET or RSET. See "MKI$, MKS$, MKD$ Functions" in this chapter.

Refer to Appendix B, "BASIC Diskette Input and Output" for a complete explanation of using random files.

Note: LSET or RSET may also be used with a string variable which was not defined in a FIELD statement to left-justify or right-justify a string in a given field. For example, the program lines:

```
110 A$=SPACE$(20)
120 RSET A$=N$
```

right-justify the string N$ in a 20-character field. This can be useful for formatting printed output.

Example: This example converts the numeric value AMT into a string, and left-justifies it in the field A$ in preparation for a PUT (file) statement.

```
150 LSET A$=MKS$(AMT)
```

MERGE Command

Purpose: Merges the lines from an ASCII program file into the program currently in memory.

Versions:

Cassette	Cartridge	Compiler
***	***	

Format: MERGE *filespec*

Remarks:

filespec is a string expression for the file specification. It must conform to the rules for naming files as outlined in "Naming Files" in Chapter 3; otherwise an error occurs and the MERGE is canceled.

The device is searched for the named file. If found, the program lines in the device file are merged with the lines in memory. If any lines in the file being merged have the same line number as lines in the program in memory, the lines from the file replace the corresponding lines in memory.

After the MERGE command, the merged program resides in memory, and BASIC returns to command level.

In Cassette BASIC, if the device name is omitted, CAS1: is assumed. CAS1: is the only allowable device for MERGE in Cassette BASIC and Cartridge BASIC when DOS is not present. With Cartridge BASIC, if the device name is omitted and DOS is present, the DOS default drive is assumed.

If CAS1: is specified as the device name and the filename is omitted, the next ASCII program file encountered on the tape is merged.

If the program being merged was not saved in ASCII format (using the **A** option on the SAVE command), a **Bad file mode** error occurs. The program in memory remains unchanged.

Example:

```
MERGE "A:NUMBRS"
```

This merges the file named "NUMBRS" on drive A with the program in memory.

MID$
Function and Statement

Purpose: Returns the requested part of a given string. When used as a statement, as in the second format, replaces a portion of one string with another string.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: As a function:

*v$* = MID$(*x$*,*n*[, *m*])

As a statement:

MID$(*v$*,*n*[, *m*]) = *y$*

Remarks: **For the function (v$=MID$...):**

x$ is any string expression.

n is an integer expression in the range 1 to 255.

m is an integer expression in the range 0 to 255.

The function returns a string of length *m* characters from *x$* beginning with the *n*th character. If *m* is omitted or if there are fewer than *m* characters to the right of the *n*th character, all rightmost characters beginning with the *n*th character are returned. If *m* is equal to zero, or if *n* is greater than LEN(*x$*), then MID$ returns a null string.

Also see the LEFT$ and RIGHT$ functions.

For the statement (MID$...=y$):

v$ is a string variable or array element that will have its characters replaced.

n is an integer expression in the range 1 to 255.

m is an integer expression in the range 0 to 255.

y$ is a string expression.

The characters in *v$*, beginning at position *n*, are replaced by the characters in *y$*. The optional *m* refers to the number of characters from *y$* that will be used in the replacement. If *m* is omitted, all of *y$* is used.

However, regardless of whether *m* is omitted or included, the length of *v$* does not change. For example, if *v$* is four characters long and *y$* is five characters long, then after the replacement *v$* will contain only the first four characters of *y$*.

> **Note:** If either *n* or *m* is out of range, an **Illegal function call** error is returned.

Example: The first example uses the MID$ function to select the middle portion of the string B$.

```
Ok
10 A$="GOOD "
20 B$="MORNING EVENING AFTERNOON"
30 PRINT A$;MID$(B$,9,7)
RUN
GOOD EVENING
Ok
```

The next example uses the MID$ statement to replace characters in the string A$.

```
Ok
10 A$="MARATHON, GREECE"
20 MID$(A$,11)="FLA.    "
30 PRINT A$
RUN
MARATHON, FLA.
Ok
```

Note in the second example how the length of A$ was not changed.

Purpose: Creates a directory on the specified diskette.

> **Note:** This command requires the use of DOS 2.10.

Versions: Cassette Cartridge Compiler

Cassette	Cartridge	Compiler
	***	

Format: MKDIR *path*

Remarks:

path is a string expression, not exceeding 63 characters, that identifies the new directory to be created. For more information about paths refer to "Naming Files" and "Tree-Structured Directories" in Chapter 3.

Examples: From the root directory, create a sub-directory called SALES.

```
MKDIR "SALES"
```

From the root directory, create a sub-directory called MIKE under the directory SALES.

```
MKDIR "SALES\MIKE"
```

From the root directory, create a sub-directory called PAM under the directory MIKE.

```
MKDIR "SALES\MIKE\PAM"
```

MKDIR Command

From the root directory, create a sub-directory called ACCOUNTING.

```
MKDIR "ACCOUNTING"
```

Make ACCOUNTING the current directory, then create two sub-directories called SHANNON and CHELLE.

```
CHDIR "ACCOUNTING":MKDIR "SHANNON":MKDIR "CHELLE"
```

The same structure could have been created from the root by entering:

```
MKDIR "ACCOUNTING\SHANNON"
MKDIR "ACCOUNTING\CHELLE"
```

By following the above examples, you have created a tree structure that looks like this:

```
              ROOT
             /    \
            /      \
        SALES    ACCOUNTING
         /        /       \
        /        /         \
    MIKE      SHANNON     CHELLE
     /
    /
  PAM
```

MKI$, MKS$, MKD$ Functions

Purpose: Convert numeric type values to string type values.

Versions:

Cassette	Cartridge	Compiler
	***	***

Format: *v$* = MKI$(*integer expression*)

*v$* = MKS$(*single-precision expression*)

*v$* = MKD$(*double-precision expression*)

Remarks: Any numeric value that is placed in a random file buffer with an LSET or RSET statement must be converted to a string. MKI$ converts an integer to a 2-byte string. MKS$ converts a single-precision number to a 4-byte string. MKD$ converts a double-precision number to an 8-byte string.

These functions differ from STR$ because they do not really change the bytes of the data, just the way BASIC interprets those bytes.

See also "CVI, CVS, CVD Functions" in this chapter and Appendix B, "BASIC Diskette Input and Output."

Example: This example uses a random file (#1) with fields defined in line 100. The first field, D$, is intended to hold a numeric value, AMT. Line 110 converts AMT to a string value using MKS$ and uses LSET to place what is really the value of AMT into the random file buffer. Line 120 places a string into the buffer (we don't need to convert a string); then line 130 writes the data from the random file buffer to the file.

```
100 FIELD #1, 4 AS D$, 20 AS N$
110 LSET D$ = MKS$(AMT)
120 LSET N$ = A$
130 PUT #1
```

Purpose: Turns the cassette player on and off from a program.

Versions:

Cassette	Cartridge	Compiler
***	***	

Format: MOTOR [*state*]

Remarks:

state is a numeric expression indicating on or off.

If *state* is non zero, the cassette motor is turned on. If *state* is zero, the cassette motor is turned off.

If *state* is omitted, the cassette motor state is switched. That is, if the motor is off, it is turned on and vice-versa.

Example: The following sequence of statements turns the cassette motor on, then off, then back on again.

```
10 MOTOR 1
20 MOTOR 0
30 MOTOR
```

NAME Command

Purpose: Changes the name of a diskette file. The NAME command in BASIC is similar to the RENAME command in DOS.

> **Note:** This command requires the use of DOS 2.10. If DOS 2.10 is not present an **Illegal function call** error will occur.

Versions:

Cassette	Cartridge	Compiler
	***	***

Format: NAME *filespec* AS *filename*

Remarks:

filespec is a file specification as outlined under "Naming Files" in Chapter 3.

filename will be the new filename. It must be a valid filename as outlined in the same section.

The file specified by *filespec* must exist and *filename* must not exist on the diskette, otherwise an error will result. If the device name is omitted, the DOS default drive is assumed. Note that the file extension does not default to **.BAS**.

After a NAME command, the file exists on the same diskette, in the same area of diskette space, with the new name.

Example:

```
NAME "A:ACCTS.BAS" AS "LEDGER.BAS"
```

In this example, the file that was formerly named ACCTS.BAS on the diskette in drive A is now named LEDGER.BAS.

NEW
Command

Purpose: Deletes the program currently in memory and clears all variables.

Versions:

Cassette	Cartridge	Compiler
***	***	

Format: NEW

Remarks: NEW is usually used to free memory before entering a new program. BASIC always returns to command level after NEW is executed. NEW causes all files to be closed, turns trace off if it was on, and resets to music background. (See "TRON and TROFF Commands," later in this chapter).

Example:

```
Ok
NEW
Ok
```

The program that had been in memory is now deleted.

Purpose: Generates noise through the external speaker.

Versions: Cassette Cartridge Compiler
*** (Cartridge)

Cassette	Cartridge	Compiler
	***	

Format: NOISE *source,volume,duration*

Remarks:

source is the noise source. It is a numeric expression in the range 0 to 7. If source is in the range 0 to 3, then periodic noise is selected. If source is in the range 4 to 7, then white noise is selected.

Periodic	White	Source is Clock Frequency (3.579)
0	4	3.579/512 (high pitch, less coarse hiss)
1	5	3.579/1024
2	6	3.579/2048 (low pitch, more coarse hiss)
3	7	source is frequency from voice 3

Note that for source values of 3 and 7, the frequency of voice 3 is used instead of the system clock. The voice 3 frequency can be set by the PLAY statement or the SOUND statement.

volume is a numeric expression in the range 0 to 15.

duration is the desired duration in clock ticks. The clock ticks occur 18.2 times per second. *duration* must be a numeric expression in the range 0 to 65535.

NOISE Statement

Sound produced by the NOISE Statement will go to the external speaker. A SOUND ON statement must be executed before using NOISE or you will get an **Illegal function call** error.

If you want to use a noise source of 3 or 7, then refer to the "PLAY Statement" and the "SOUND Statement" in this chapter for information on multiple voices.

Example:

```
10  SOUND ON
20  FOR N=0 TO 7
30  NOISE N,15,250
40  PLAY "","","V0"
50  FOR I=1 to 6
60  PLAY "","","V15;O=I;CDEF"
70  NEXT I
80  NEXT N
```

This example demonstrates all the possible noise sources. Note the use of three voices with the play statement.

Purpose: Returns a string which represents the octal value of the decimal argument.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: *v$* = OCT$(*n*)

Remarks:

n is a numeric expression in the range -32768 to 65535.

If *n* is negative, the two's complement form is used. That is, OCT$(-*n*) is the same as OCT$(65536-*n*).

See the HEX$ function for hexadecimal conversion.

Example:

```
Ok
PRINT OCT$(24)
30
Ok
```

This example shows that 24 in decimal is 30 in octal.

ON COM(n) Statement

Purpose: Sets up a line number for BASIC to trap to when there is information coming into the communications buffer.

Versions:

Cassette	Cartridge	Compiler
	***	(**)

Format: ON COM(*n*) GOSUB *line*

Remarks:

n is the number of the communications ports (1 or 2).

line is the line number of the beginning of the trap routine. Setting *line* equal to 0 (zero) disables trapping of communications activity for the specified port.

A COM(*n*) ON statement must be executed to activate this statement for port *n*. After COM(*n*) ON, if a non-zero line number is specified in the ON COM(*n*) statement then every time the program starts a new statement, BASIC checks to see if any characters have come in to the specified communications port. If so, BASIC performs a GOSUB to the specified *line*.

If COM(*n*) OFF is executed, no trapping takes place for the port. Even if communications activity does take place, the event is not remembered.

If a COM(*n*) STOP statement is executed, no trapping takes place for the port. However, any characters being received are remembered so an immediate trap takes place when COM(*n*) ON is executed.

When the trap occurs an automatic COM(*n*) STOP is executed so recursive traps can never take place. The RETURN from the trap routine automatically does a COM(*n*) ON unless an explicit COM(*n*) OFF was performed inside the trap routine.

Event trapping does not take place when BASIC is not executing a program. When an error trap (resulting from an ON ERROR statement) takes place all trapping is automatically disabled (including ERROR, STRIG(n), PEN, COM(n), and KEY(n)).

Typically, the communications trap routine reads an entire message from the communications line before returning back. It is not recommended that you use the communications trap for single character messages since at high baud rates the overhead of trapping and reading for each individual character may allow the interrupt buffer for communications to overflow.

You may use RETURN *line* if you want to go back to the BASIC program at a fixed line number. Use of this non-local return must be done with care, however, since any other GOSUBs, WHILEs, or FORs that were active at the time of the trap will remain active.

ON COM(n) Statement

Example:

```
150 ON COM(1) GOSUB 500
160 COM(1) ON
.
.
.
500 REM incoming characters
.
.
.
590 RETURN 300
```

This example sets up a trap routine for communications at line 500.

Purpose: Enables error trapping and specifies the first line of the error handling subroutine.

Versions:

Cassette	Cartridge	Compiler
***	***	(**)

Format: ON ERROR GOTO *line*

Remarks:

line is the line number of the first line of the error trapping routine. If the line number does not exist, an **Undefined line number** error results.

Once error trapping has been enabled, all errors detected (*including direct mode errors*) will cause a jump to the specified error handling subroutine.

To disable error trapping, execute an ON ERROR GOTO 0. Subsequent errors will print an error message and halt execution. An ON ERROR GOTO 0 statement that appears in an error trapping subroutine causes BASIC to stop and print the error message for the error that caused the trap. It is recommended that all error trapping subroutines execute an ON ERROR GOTO 0 if an error is encountered for which there is no recovery action.

BASIC considers itself to be within the error trapping routine from the time an error occurs and it branches to the line specified by the ON ERROR statement, until a RESUME statement is encountered. You use the RESUME statement to exit from the error trapping routine; a simple GOTO statement is not sufficient. Refer to "RESUME Statement" in this chapter. Because error trapping does not occur within the error trapping routine, an

ON ERROR
Statement

ON ERROR GOTO *line* (within the error trapping routine), where *line* is anything other than 0, will not work.

> **Note:** If an error occurs during execution of an error handling subroutine, the BASIC error message is printed and execution terminates. Error trapping does not occur within the error handling subroutine.

Example:

```
10 ON ERROR GOTO 100
20 LPRINT "This goes to the printer."
30 END
100 IF ERR=27 THEN PRINT "Check printer"
    : RESUME
```

This example shows how you might trap a common error—forgetting to put paper in the printer, or forgetting to switch it on.

ON-GOSUB and ON-GOTO Statements

Purpose: Branches to one of several specified line numbers, depending on the value of an expression.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: ON *n* GOTO *line*[,*line*]...

ON *n* GOSUB *line*[,*line*]...

Remarks:

n is a numeric expression which is rounded to an integer, if necessary. It must be in the range 0 to 255, or an **Illegal function call** error occurs.

line is the line number of a line you wish to branch to.

The value of *n* determines which line number in the list will be used for branching. For example, if the value of *n* is 3, the third line number in the list will be the destination of the branch.

In the ON–GOSUB statement, each line number in the list must be the first line number of a subroutine. That is, you eventually need to have a RETURN statement to bring you back to the line following the ON–GOSUB.

If the value of *n* is zero or greater than the number of items in the list (but less than or equal to 255), BASIC continues with the next executable statement.

ON-GOSUB and ON-GOTO Statements

Example: The first example branches to line 150 if L-1 equals 1, to line 300 if L-1 equals 2, to line 320 if L-1 equals 3, and to line 390 if L-1 equals 4. If L-1 is equal to 0 (zero) or is greater than 4, then the program just goes on to the next statement.

```
100 ON L-1 GOTO 150,300,320,390
```

The next example shows how to use an ON–GOSUB statement.

```
100 REM display menu
110 PRINT "1.  Routine 1"
120 PRINT "2.  Routine 2"
130 PRINT "3.  Routine 3"
140 PRINT "4.  Routine 4"
150 INPUT "Your choice?"; CHOICE
160 ON CHOICE GOSUB 200, 300, 400, 500
170 GOTO 100 ' redisplay menu after routine is done
200 REM start of first routine
.
.
.
290 RETURN
300 REM start of second routine
.
.
.
```

Purpose: Sets up a line number for BASIC to trap to when the specified function key or cursor control key is pressed.

Versions:

Cassette	Cartridge	Compiler
	***	(**)

Format: ON KEY(*n*) GOSUB *line*

Remarks:

n is a numeric expression in the range 1 to 20 indicating the key to be trapped, as follows:

1-10	function keys Fn and F1 to F10
11	Cursor Up
12	Cursor Left
13	Cursor Right
14	Cursor Down
15-20	keys defined by the form: KEY *n*,CHR$(*shift*)+CHR$(*scan code*). See "KEY Statement" and "KEY(n) Statement" in this chapter for more information. This form is not supported in the BASIC Compiler.

line is the line number of the beginning of the trapping routine for the specified key. Setting *line* equal to 0 (zero) stops trapping of the key.

A KEY(*n*) ON statement must be executed to activate this statement. After KEY(*n*) ON, if a non-zero line number is specified in the ON KEY(*n*) statement then every time the program starts a new

STATEMENTS

statement, BASIC checks to see if the specified key was pressed. If so, BASIC performs a GOSUB to the specified *line*.

If a KEY(*n*) OFF statement is executed, no trapping takes place for the specified key. Even if the key is pressed, the event is not remembered.

If a KEY(*n*) STOP statement is executed, no trapping takes place for the specified key. However, if the key is pressed the event is remembered, so an immediate trap takes place when KEY(*n*) ON is executed.

When the trap occurs an automatic KEY(*n*) STOP is executed so recursive traps can never take place. The RETURN from the trap routine automatically does a KEY(*n*) ON unless an explicit KEY(*n*) OFF was performed inside the trap routine.

Event trapping does not take place when BASIC is not executing a program. When an error trap (resulting from an ON ERROR statement) takes place all trapping is automatically disabled (including ERROR, STRIG(*n*), PEN, COM(*n*), and KEY(*n*)).

Key trapping may not work when other keys are pressed before the specified key. The key that caused the trap cannot be tested using INPUT$ or INKEY$, so the trap routine for each key must be different if a different function is desired.

You may use RETURN *line* if you want to go back to the BASIC program at a fixed line number. Use of this non-local return must be done with care, however, since any other GOSUBs, WHILEs, or FORs that were active at the time of the trap will remain active.

KEY(*n*) ON has no effect on whether the soft key values are displayed at the bottom of the screen.

Example: The following is an example of a trap routine for function key 5.

```
100 ON KEY(5) GOSUB 200
110 KEY(5) ON
.
.
.
200 REM function key 5 pressed
.
.
.
290 RETURN 140
```

ON KEY(n) Statement

This example traps Fn/Break and Ctrl-Alt-Del. Note that this example only works in Cartridge BASIC.

```
10 KEY 15,CHR$(&H04)+CHR$(70) ' Fn/Break
20 KEY 16,CHR$(&H04+&H08)+CHR$(83)
     'Ctrl-Alt-Del
30 ON KEY (15) GOSUB 1000
40 ON KEY (16) GOSUB 2000
50 KEY (15) ON: KEY (16) ON
.
.
.
1000 PRINT "Trapping for Fn and Break"
1010 RETURN
2000 TRAPS=TRAPS+1
2010 ON TRAPS GOTO 2100,2200,2300,2400,2500
2020 '
2100 PRINT "1st trap of System Reset":RETURN
2200 PRINT "2nd trap of System Reset":RETURN
2300 PRINT "3rd trap of System Reset":RETURN
2400 PRINT "4th trap of System Reset":RETURN
2500 KEY (16) OFF 'Disable trap of Sys. Reset
2510 RETURN
```

Purpose: Sets up a line number for BASIC to transfer control to when the light pen is activated.

Versions:

Cassette	Cartridge	Compiler
	***	(**)

Format: ON PEN GOSUB *line*

Remarks:

line is the line number of the beginning of the trap routine for the light pen. Using a line number of 0 disables trapping of the light pen.

A PEN ON statement must be executed to activate this statement. After PEN ON, if a non-zero line number is specified in the ON PEN statement, then every time the program starts a new statement BASIC will check to see if the pen was activated. If so, BASIC performs a GOSUB *line*.

If PEN OFF is executed, no trapping takes place. Even if the light pen is activated, the event is not remembered.

If a PEN STOP statement is executed, no trapping takes place, but pen activity is remembered so that an immediate trap takes place when PEN ON is executed.

When the trap occurs, an automatic PEN STOP is executed so recursive traps can never take place. The RETURN from the trap routine automatically does a PEN ON unless an explicit PEN OFF was performed inside the trap routine.

ON PEN Statement

Event trapping does not take place when BASIC is not executing a program. When an error trap (resulting from an ON ERROR statement) takes place all trapping is automatically disabled (including ERROR, STRIG(*n*), PEN, COM(*n*), and KEY(*n*)).

PEN(0) is not set when pen activity causes a trap.

You may use RETURN *line* if you want to go back to the BASIC program at a fixed line number. Use of this non-local return must be done with care, however, since any other GOSUBs, WHILEs, or FORs that were active at the time of the trap will remain active.

> **Note:** Do not try any cassette I/O while PEN is ON.

Example: This example sets up a trap routine for the light pen.

```
10 ON PEN GOSUB 500
20 PEN ON
.
.
.
500 REM subroutine for pen
.
.
.
650 RETURN 30
```

ON PLAY(n) Statement

Purpose: Allows continuous music to play during program execution.

Versions: Cassette Cartridge Compiler
*** (Cartridge)

Cassette	Cartridge	Compiler
	***	

Format: ON PLAY(*n*) GOSUB *line*

Remarks:

n is an integer expression in the range 1 to 32 indicating the notes to be trapped. Values entered outside of this range result in an **Illegal function call** error.

line is the beginning line number of the trap routine for PLAY. A line number of 0 (zero) stops play trapping.

A PLAY ON statement must be used to start the ON PLAY(n) statement. After PLAY ON, if a non-zero line number is specified in the PLAY(*n*) statement, each time the program starts a new statement BASIC checks to see if the music buffer has gone from *n* to *n*-1 notes. If so, BASIC performs a GOSUB to the specified line.

If multiple voices are playing simultaneously the last voice to go from *n* to *n*-1 in the queue will cause BASIC to perform the GOSUB to the specified line.

If PLAY OFF is used, no trapping takes place. Even if a play activity takes place, the event is not remembered.

STATEMENTS

If a PLAY STOP statement is used, no trapping takes place, but play activity is remembered so that an immediate trap takes place when PLAY ON is executed.

When the trap occurs, an automatic PLAY STOP is run so recursive traps can never take place. The RETURN from the trap routine automatically does a PLAY ON unless an explicit PLAY OFF was performed inside the trap routine.

Event trapping does not take place when BASIC is not running a program. When an error trap (resulting from an ON ERROR statement) takes place all trapping is automatically disabled (including ERROR, STRIG(*n*), PEN, COM(*n*), and KEY(*n*)).

You can use RETURN *line* if you want to go back to the BASIC program at a fixed line number. You must use this non-local return with care, however, since any other GOSUBs, WHILEs, or FORs that were active at the time of the trap will remain active.

Notes:

1. A PLAY event trap is not issued if the music buffer is already empty when a PLAY ON statement is performed.

2. Be careful choosing values for *n*. For example: ON PLAY(32) causes so many event traps that little time remains to run the rest of the program.

Refer to the "PLAY(n) Function" in this chapter for additional information.

Example: This example sets up a trap routine which is invoked when five notes are left in the background music buffer.

```
10 ON PLAY(5) GOSUB 500
20 PLAY ON
.
.
.
500 REM subroutine for background music
.
.
.
650 RETURN 30
```

ON STRIG(n) Statement

Purpose: Sets up a line number for BASIC to trap to when one of the joystick buttons (triggers) is pressed.

Versions:

Cassette	Cartridge	Compiler
	***	(**)

Format: ON STRIG(*n*) GOSUB *line*

Remarks:

n may be 0, 2, 4, or 6, and indicates the button to be trapped as follows:

0	button A1
2	button B1
4	button A2
6	button B2

line is the beginning line number of the trap routine for STRIG. A line number of 0 (zero) stops trapping of the joystick button.

A STRIG(*n*) ON statement must be executed to activate this statement for button *n*. If STRIG(*n*) ON is executed and a non-zero line number is specified in the ON STRIG(*n*) statement, then every time the program starts a new statement BASIC checks to see if the specified button has been pressed. If so, BASIC performs a GOSUB to the specified *line*.

If STRIG(*n*) OFF is executed, no trapping takes place for button *n*. Even if the button is pressed, the event is not remembered.

If a STRIG(*n*) STOP statement is executed, no trapping takes place for button *n*, but the button being pressed is remembered so that an immediate trap takes place when STRIG(*n*) ON is executed.

When the trap occurs, an automatic STRIG(*n*) STOP is executed so recursive traps can never take place. The RETURN from the trap routine automatically does a STRIG(*n*) ON unless an explicit STRIG(*n*) OFF was performed inside the trap routine.

Event trapping does not take place when BASIC is not executing a program. When an error trap (resulting from an ON ERROR statement) takes place all trapping is automatically disabled (including ERROR, STRIG(*n*), PEN, COM(*n*), and KEY(*n*)).

Using STRIG(*n*) ON will activate the interrupt routine that checks the button status for the specified joy stick button. Downstrokes that cause trapping will not set functions STRIG(0), STRIG(2), STRIG(4), or STRIG(6).

You may use RETURN *line* if you want to go back to the BASIC program at a fixed line number. Use of this non-local return must be done with care, however, since any other GOSUBs, WHILEs, or FORs that were active at the time of the trap will remain active.

ON STRIG(n) Statement

Example: This is an example of a trapping routine for the button on the first joy stick.

```
100 ON STRIG(0) GOSUB 2000
110 STRIG(0) ON
.
.
.
2000 REM subroutine for 1st button
.
.
.
2100 RETURN
```

ON TIMER Statement

Purpose: Transfers control to a given line number in a BASIC program when a defined period of time has elapsed.

> **Note:** This statement requires the use of DOS 2.10.

Versions:

Cassette	Cartridge	Compiler
	***	

Format: ON TIMER(*n*) GOSUB *line*

Remarks:

n is a numeric expression in the range 1 to 86,400 (1 second through 24 hours). Values entered that are outside of this range result in an **Illegal function call** error.

line is the beginning line number of the trap routine for TIMER. A line number of 0 (zero) stops timer trapping.

A TIMER ON statement must be used to start the ON TIMER statement. After TIMER ON, specifying a non-zero line number in the ON TIMER(*n*) statement causes BASIC to keep track of the passing seconds. When *n* seconds have elapsed, BASIC performs a GOSUB to the specified line. The event trap occurs, and BASIC starts counting again from 0.

If TIMER OFF is used, no trapping takes place. Even if TIMER activity takes place, the event is not remembered.

ON TIMER
Statement

If a TIMER STOP statement is used, no trapping takes place, but TIMER activity is remembered so that an immediate trap occurs when TIMER ON is used.

When the trap occurs, an automatic TIMER STOP is executed so recursive traps can never take place. The RETURN from the trap routine automatically does a TIMER ON unless an explicit TIMER OFF was performed inside the trap routine.

Event trapping does not take place when BASIC is not running a program. When an error trap (resulting from an ON ERROR statement) takes place all trapping is automatically disabled (including ERROR, STRIG(*n*), PEN, COM(*n*), KEY(*n*) and PLAY).

You can use RETURN *line* if you want to go back to the BASIC program at a fixed line number. You must use this non-local return with care, however, since any other GOSUBs, WHILEs, or FORs that were active at the time of the trap will remain active.

ON TIMER is useful in programs that need an interval timer. For example, to display the time of day on line one every minute:

```
10 ON TIMER(60) GOSUB 10000
20 TIMER ON
.
.
.
10000 OLDROW=CSRLIN 'save current row
10010 OLDCOL=POS(0) 'save current column
10020 LOCATE 1,1: PRINT TIME$;
10030 LOCATE OLDROW,OLDCOL 'restore row,col
10040 RETURN
```

Purpose: Allows I/O to a file or device.

Versions:

Cassette	Cartridge	Compiler
***	***	(**)

Format: First form:

OPEN *filespec* [FOR *mode*] AS [#]*filenum* [LEN=*recl*]

OPEN *path* [FOR *mode*] AS [#]*filenum* [LEN=*recl*]

Alternate form:

OPEN *mode2*, [#]*filenum, filespec* [,*recl*]

OPEN *mode2*, [#]*filenum, path* [,*recl*]

Remarks:

mode in the first form, is one of the following:

OUTPUT specifies sequential output mode.

INPUT specifies sequential input mode.

APPEND specifies sequential output mode where the file is positioned to the end of data on the file when it is opened.

Note that *mode* must be a string constant, *not* enclosed in quotation marks. If *mode* is omitted, random access is assumed.

mode2 in the alternate form, is a string expression with the first character being one of the following:

O specifies sequential output mode

I specifies sequential input mode

R specifies random input/output mode

For both formats:

filenum is an integer expression whose value is between one and the maximum number of files allowed. In Cassette BASIC and in Cartridge BASIC without DOS the maximum number is 4. In Cartridge BASIC, when DOS is present, the default maximum is 3, but this can be changed with the **/F:** switch on the BASIC command line.

filespec is a string expression for the file specification as explained under "Naming Files" in Chapter 3.

path is a string expression not exceeding 63 characters as explained under "Naming Files" in Chapter 3. Refer also to "Tree-Structured Directories" in Chapter 3. Valid only in Cartridge BASIC when using DOS 2.0.

recl is an integer expression which, if included, sets the record length for random files. It may range from 1 to 32767. In Cartridge BASIC, you can use *recl* for sequential

files. The default record length is 128 bytes. *recl* may not exceed the value set by the **/S:** switch on the BASIC command.

OPEN allocates a buffer for I/O to the file or device and determines the mode of access that will be used with the buffer.

filenum is the number that is associated with the file for as long as it is open and is used by other I/O statements to refer to the file or device.

An OPEN must be executed before any I/O may be done to a device or file using any of the following statements, or any statement or function requiring a file number:

PRINT #	INPUT #
PRINT # USING	LINE INPUT #
WRITE #	GET
INPUT$	PUT

GET and PUT are valid for random files (or communications file—see the "OPEN "COM... Statement" section). A diskette file may be either random or sequential, and a printer may be opened in either random or sequential mode; however, all other devices may be opened only for sequential operations.

BASIC normally adds a line feed after each carriage return (CHR$(13)) sent to a printer. However, if you open a printer (LPT1:) as a random file with width 255, this line feed is suppressed.

APPEND is valid only for diskette files. The file pointer is initially set to the end of the file and the record number is set to the last record of the file. PRINT # or WRITE # will then extend the file.

Note: At any one time, it is possible to have a particular file open under more than one file number. This allows different modes to be used for different purposes. Or, for program clarity, you may use different file numbers for different modes of access. Each file number has a different buffer, so you should use care if you are writing using one file number and reading using another file number.

However, a file cannot be opened for sequential output or append if the file is already open.

If the device name is omitted when you are using Cassette BASIC, CAS1: is assumed. If you are using Cartridge BASIC and DOS is present. the DOS default drive is assumed.

If CAS1: is specified as the device and the filename is omitted, then the next data file on the cassette is opened.

In Cassette BASIC, and Cartridge BASIC without DOS, a maximum of four files may be open at one time (cassette, printer, keyboard, and screen). Note that only one cassette file may be open at a time. For Cartridge BASIC the default maximum is three files when DOS is present. If using DOS, you can override this value by using the **/F:** option on the BASIC command line.

If a file opened for input does not exist, a **File not found** error occurs. If a file which does not exist is opened for output, append, or random access, a file is created.

Any values given outside the ranges indicated will result in an **Illegal function call** error. The file is not opened.

See Appendix B, "BASIC Diskette Input and Output" for a complete explanation of using diskette files. Refer to the next section, "OPEN "COM... Statement," for information on opening communications files.

Examples: Either of these statements opens the file named "DATA" for sequential output on the default device (CAS1: for Cassette BASIC, default drive for Cartridge BASIC when using DOS).

```
10 OPEN "DATA" FOR OUTPUT AS #1
```
or
```
10 OPEN "O",#1,"DATA"
```

In the above example, note that opening for output destroys any existing data in the file. If you do not wish to destroy data you should open for APPEND.

```
20 OPEN "A:SSFILE" AS 1 LEN=256
```
or
```
20 OPEN "R",1,"A:SSFILE",256
```

Either of the preceding two statements opens the file named "SSFILE" on the diskette in drive A for random input and output. The record length is 256.

```
25 FILE$ = "A:DATA.ART"
30 OPEN FILE$ FOR APPEND AS 3
```

This example opens the file "DATA.ART" on the diskette in drive A and positions the file pointers so that any output to the file is placed at the end of existing data in the file.

OPEN Statement

```
Ok
10 OPEN "LPT1:" AS #1' random access
20 PRINT #1,"Printing width 80"
30 PRINT #1,"Now change to width 255"
40 WIDTH #1,255
50 PRINT #1,"This line will be underlined"
60 WIDTH #1,80
70 PRINT #1, STRING$(28,"_")
80 PRINT #1,"Printing width 80 with CR/LF"
RUN
OK
```

This is printed on the printer:

```
Printing width 80
Now change to width 255
This line will be underlined
Printing width 80 with CR/LF
```

Line 10 in this example opens the printer in random mode. Because the default width is 80, the lines printed by lines 20 and 30 end with a carriage return/line feed. Line 40 changes the printer width to 255, so the line feed after the carriage return is suppressed. Therefore, the line printed by line 50 ends only with a carriage return and not a line feed. This causes the line printed by line 70 to overprint "This line will be underlined," causing the line to be underlined. Line 60 changes the width back to 80 so the underlines and following lines will end with a line feed.

In Cartridge BASIC, when DOS is present, it is possible to OPEN files using paths as described under "Naming Files" in Chapter 3. The following examples illustrate the use of paths for *filespec*.

```
10 OPEN "LVL1 LVL2 DATA" FOR OUTPUT AS #1
```
or
```
10 OPEN "O",#1,"LVL1 LVL2 DATA"
```

Either of these statements opens the file called "DATA" for sequential output on the default device in the directory called LVL2.

```
20 OPEN "A:LVL1 RRFILE" AS 1 LEN=256
```
or
```
20 OPEN "R",1,"A:LVL1 RRFILE",256
```

Either of the preceding two statements opens the file named "RRFILE" in the LVL1 directory on the diskette in drive A for random input and output. The record length is 256.

```
25 FILE$="A:LVL1 LVL2 LVL3 DATA.FIL"
30 OPEN FILE$ FOR APPEND AS 3
```

This example opens the file "DATA.FIL" on the diskette in drive A in the directory called LVL3 and positions the file pointers so that any output to the file is placed at the end of the existing data in the file.

OPEN "COM... Statement

Purpose: Opens a communications file.

Versions: Cassette | Cartridge *** | Compiler (**)

Format: OPEN "COM*n*:[*speed*] [, *parity*] [, *data*] [, *stop*] [,RS] [,CS[*n*]] [,DS[*n*]] [,CD[*n*]] [,LF] [,PE]" AS [#]*filenum* [LEN=*number*]

Remarks:

n is 1 or 2, indicating the number of the Asynchronous Communications port.

speed is an integer constant specifying the transmit/receive bit rate in bits per second (bps). Valid speeds are 75, 110, 150, 300, 600, 1200, 1800, 2400, and 4800 for the RS232 Serial Port. Valid speeds for the IBM Modem option are 110 or 300. The default is 300 bps.

parity is a one-character constant specifying the parity for transmit and receive as follows:

S SPACE: Parity bit always transmitted and received as a space (0 bit).

O ODD: Odd transmit parity, odd receive parity checking.

M MARK: Parity bit always transmitted and received as a mark (1 bit).

E EVEN: Even transmit parity, even receive parity checking.

N NONE: No transmit parity, no receive parity checking.

The default is EVEN (E).

data is an integer constant indicating the number of transmit/receive data bits. Valid values are: 4, 5, 6, 7, or 8. The default is 7.

stop is an integer constant indicating the number of stop bits. Valid values are 1 or 2. The default is two stop bits for 75 and 110 bps, one stop bit for all others. If you use 4 or 5 for *data*, a 2 here will mean 1 1/2 stop bits.

filenum is an integer expression which evaluates to a valid file number. The number is then associated with the file for as long as it is open and is used by other communications I/O statements to refer to the file.

number is the maximum number of bytes which can be read from the communication buffer when using GET or PUT. The default is 128 bytes.

OPEN "COM... allocates a buffer for I/O in the same fashion as OPEN for diskette files. It supports RS232 asynchronous communication with other computers and peripherals.

Note: If the keyboard is struck or the diskette drive is read from or written to, then characters

may be lost unless you use XOFF to stop communication transmission. See the example in Appendix F.

A communications device may be open to only one file number at a time.

The **RS, CS, DS, CD, LF** and **PE** options affect the line signals as follows:

RS	suppresses RTS (Request To Send)
CS[n]	controls CTS (Clear To Send)
DS[n]	controls DSR (Data Set Ready)
CD[n]	controls CD (Carrier Detect)
LF	sends a line feed following each carriage return
PE	enables parity checking

The CD (Carrier Detect) is also known as the RLSD (Received Line Signal Detect).

> **Note:** The *speed, parity, data*, and *stop* parameters are positional, but **RS, CS, DS, CD, LF**, and **PE** are not.

The RTS (Request To Send) line is turned on when you execute an OPEN "COM... statement unless you include the **RS** option.

The *n* argument in the **CS, DS**, and **CD** options specifies the number of milliseconds to wait for the signal before returning a **Device Timeout** error. *n* may range from 0 to 65535. If *n* is omitted or is equal to zero, then the line status is not checked at all.

The defaults are **CS1000, DS1000**, and **CD0**. If **RS** was specified, **CS0** is the default.

That is, normally I/O statements to a communications file will fail if the CTS (Clear To Send) or DSR (Data Set Ready) signals are off. The system waits one second before returning a **Device Timeout**. The **CS** and **DS** options allow you to ignore these lines or to specify the amount of time to wait before the timeout.

Normally Carrier Detect (CD or RLSD) is ignored when an OPEN "COM... statement is executed. The **CD** option allows you to test this line by including the *n* parameter, in the same way as **CS** and **DS**. If *n* is omitted or is equal to zero, then Carrier Detect is not checked at all (which is the same as omitting the **CD** option).

The **LF** parameter is intended for those using communication files as a means of printing to a serial line printer. When you specify **LF**, a line feed character (hex 0A) is automatically sent after each carriage return character (hex 0C). (This includes the carriage return sent as a result of the width setting.) Note that INPUT # and LINE INPUT #, when used to read from a communications file that was opened with the **LF** option, stop when they see a carriage return. The line feed is always ignored.

The **PE** option enables parity checking. The default is no parity checking. The **PE** option will cause a Device I/O error on parity errors and will turn the high order bit on for 7 or less data bits. The **PE** option does *not* affect framing and overrun errors. These errors will always turn on the high order bit and cause a **Device I/O** error.

Any coding errors within the string expression starting with *speed* result in a **Bad file name** error. An indication as to which parameter is in error is not given.

OPEN "COM...
Statement

Refer to Appendix F, "Communications," for more information on control of output signals and other technical information on communications support.

If you specify 8 data bits, you must specify parity N. If you specify 4 data bits, you must specify a parity, that is, N parity is invalid. BASIC uses all 8 bits in a byte to store numbers, so if you are transmitting or receiving numeric data (for example, by using PUT), you must specify 8 data bits. (This is not so if you are sending numeric data *as text.*)

Refer to the previous section for opening devices other than communications devices.

Examples:

```
10 OPEN "COM1:" AS 1
```

File 1 is opened for communication with all defaults. The speed is 300 bps with even parity. There will be 7 data bits and one stop bit.

```
10 OPEN "COM1:2400" AS #2
```

File 2 is opened for communication at 2400 bps. Parity, number of data bits, and number of stop bits are defaulted.

```
20 OPEN "COM2:1200,N,8" AS #1
```

File number 1 is opened for asynchronous I/O at 1200 bps, no parity is to be produced or checked, 8-bit bytes will be sent and received, and 1 stop bit will be transmitted.

```
10 OPEN "COM1:300,N,8,,CS,DS,CD" AS #1
```

Opens COM1 at 300 bps with no parity and eight data bits. CTS, DSR, and RLSD are not checked.

```
50 OPEN "COM1:1200,,,,CS,DS2000" AS #1
```

Opens COM1 at 1200 bps with the defaults of even parity and seven data bits. RTS is sent, CTS is not checked, and **Device Timeout** is given if DSR is not seen within two seconds. Note that the commas are required to indicate the position of the *parity, start*, and *stop* parameters, even though a value is not specified. This is what is meant by *positional* parameters.

An OPEN statement may be used with an ON ERROR statement to make sure a modem is working properly before sending any data. For example, the following program makes sure we get Carrier Detect (CD or RLSD) from the modem before starting. Line 20 is set to timeout after 10 seconds. TRIES is set to 6 so we give up if Carrier Detect is not seen within one minute. Once communication is established, we reopen the file with a shorter delay until timeout.

```
5 TRIES=6
10 ON ERROR GOTO 100
20 OPEN "COM1:300,N,8,2,CS,DS,CD10000" AS #1
30 ON ERROR GOTO 0
40 CLOSE #1 ' works so can continue
50 GOTO 1000
.
.
.
100 TRIES=TRIES-1
110 IF TRIES=0 THEN ON ERROR GOTO 0 ' give up
120 RESUME
.
.
.
1000 OPEN "COM1:300,N,8,2,CS,DS,CD2000" AS #1
```

The next example shows a typical way to use a communication file to control a serial line printer. The **LF** parameter in the OPEN statement ensures that lines do not print on top of each other.

```
10 WIDTH "COM1:", 132
20 OPEN "COM1:1200,N,8,,CS10000,DS10000,CD10000,LF"
      AS #1
```

OPTION BASE Statement

Purpose: Declares the minimum value for array subscripts.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: OPTION BASE *n*

Remarks: *n* is 1 or 0.

The default base is 0. If the statement:

```
OPTION BASE 1
```

is executed, the lowest value an array subscript may have is one.

The OPTION BASE statement must be coded *before* you define or use any arrays. An error occurs if you change the base value when arrays exist.

OUT
Statement

Purpose: Sends a byte to a machine output port.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: OUT *n,m*

Remarks:

n is a numeric expression for the port number, in the range of 0-65535.

m is a numeric expression for the data to be transmitted, in the range of 0-255.

Refer to the PCjr *Technical Reference* manual for a description of valid port numbers (I/O addresses).

OUT is the complementary statement to the INP function. Refer to "INP Function" in this chapter.

One use of OUT is to affect the video output. On some displays you may find that the first two or three characters on the line don't show up on the screen. If your display does not have a horizontal adjustment control, you can use the following statements to shift the display in 40 column width:

```
OUT 980,2: OUT 981,41
```

This shifts the display two characters to the right. The shift caused by this OUT statement remains in effect until a WIDTH or SCREEN statement is executed. Another way to shift the screen is with the key combination Alt-Ctrl-← or Alt-Ctrl-→.

Example:

```
100 OUT 32,100
```

This sends the value 100 to output port 32.

PAINT
Statement

Purpose: Fills in an area on the screen with the selected color.

Versions:

Cassette	Cartridge	Compiler
	***	(**)

Graphics mode only.

Format: PAINT *(x,y)* [[, *paint*][, *boundary*][, *background*]]

Remarks:

(x,y) are the coordinates of a point within the area to be filled in. The coordinates may be given in absolute or relative form (see "Specifying Coordinates" under "Graphics Modes" in Chapter 3). This point will be used as a starting point.

paint can be a numeric or string expression. If *paint* is a numeric expression, it is the attribute to be painted with. If *paint* is omitted, it defaults to the maximum attribute for the current screen mode. In low resolution, there are 16 attributes available (0 to 15). In medium resolution, there are 4 (0 to 3) or 16 (0 to 15) attributes available, depending on the current screen mode. In high resolution, there are 2 (0 to 1) or 4 (0 to 3) attributes available, depending on the current screen mode. For more information see "Graphics Modes" in Chapter 3.

If *paint* is a string expression, then "tiling" is performed, as described later in this section.

boundary is the edge of the figure to be filled, in the range 0 to 3 or 0 to 15, depending on the screen mode, as described above. If *boundary* is omitted, then the paint attribute is assumed.

background is a one-byte string expression used in paint tiling.

The figure to be filled in is the figure with edges of *boundary* color. That is, the figure should be drawn in the *boundary* color. The figure is filled in with the color *paint*.

In medium resolution, we can fill in a border of color 2 with color 1. Visually this might mean a green ball with a red border.

The starting point of PAINT must be inside the figure to be painted. Points plotted that are outside of the screen limits are not drawn and no error occurs. If the specified point already has the color *boundary*, then PAINT will have no effect. If *paint* is omitted the maximum attribute of the current screen mode is used (15 in low resolution, 15 or 3 in medium resolution, and 1 or 3 in high resolution). PAINT can paint any type of figure; "jagged" edges on a figure will increase the amount of stack space required by PAINT. So if a lot of complex painting is being done you may want to use CLEAR at the beginning of the program to increase the stack space available. Getting an **Out of memory** error on a PAINT statement is a good indication that more stack space is required.

When *paint* is a string expression, painting can be performed as a pattern of 1 or more colors. Examples of the form of the string expression follow later in this section.

Occasionally, you may want to tile paint over an already painted area that is the same color as two consecutive lines in the tile pattern. Normally, this constitutes a terminating condition because your point is surrounded by the same bit pattern that is being plotted.

You can use the *background* attribute to skip this terminating condition. You cannot specify more than two consecutive lines in the tile pattern matching this *background* attribute. Specifying more than two consecutive lines in the tile pattern that match *background* causes an **Illegal function call** error.

The PAINT statement allows scenes to be displayed with very few statements.

The PAINT statement in line 20 fills in the box drawn in line 10 with color number 1.

```
5 SCREEN 1
10 LINE (0,0)-(100,150),2,B
20 PAINT (50,50),1,2
```

To use paint tiling, the *paint* attribute must be a string expression of the form:

CHR$(&Hnn)+CHR$(&Hnn)+CHR$(&Hnn)

The tile mask is always 8 bits wide. The two hexadecimal numbers in the CHR$ expression correspond to 8 bits. The string expression may contain up to 64 bytes. The structure of the string expression appears as follows:

	x increases --> bit of Tile byte	
x,y	7 6 5 4 3 2 1 0	
0,0	x x x x x x x x	Tile byte 0
0,1	x x x x x x x x	Tile byte 1
0,2	x x x x x x x x	Tile byte 2
.		
.		
0,63	x x x x x x x x	Tile byte 63 (maximum allowed)

The tile pattern is repeated uniformly over the entire screen. Each byte in the tile string masks 8 bits along the **x** axis when plotting points. Each byte in the tile string is rotated as required to align along the y axis, such that **tile.byte.mask=y mod tile.length**.

Because there is only one bit per pixel in high resolution (screen 2) a point is plotted at every position in the bit mask which has a value of 1. The screen can be painted with x's using the following example:


```
PAINT (320,100),CHR$(&H81)+CHR$(&H42)+CHR$(&H24)
+CHR$(&H18)+CHR$(&H18)+CHR$(&H24)+CHR$(&H42)+
CHR$(&H81)
```

The length of this mask is eight, indexed zero through seven. In this case, PAINT at coordinates (320,100) will begin by plotting byte four. This is calculated using the **y mod tile.length** formula by substituting 100 for **y** and eight for **tile.length**. This pattern appears on the screen as:

	x increases --> bit of Tile byte	
	7 6 5 4 3 2 1 0	
Tile byte 0	1 0 0 0 0 0 0 1	CHR$(&H81)

Tile byte 1	0 1 0 0 0 0 1 0	CHR$(&H42)
Tile byte 2	0 0 1 0 0 1 0 0	CHR$(&H24)
Tile byte 3	0 0 0 1 1 0 0 0	CHR$(&H18)
Tile byte 4	0 0 0 1 1 0 0 0	CHR$(&H18)
Tile byte 5	0 0 1 0 0 1 0 0	CHR$(&H24)
Tile byte 6	0 1 0 0 0 0 1 0	CHR$(&H42)
Tile byte 7	1 0 0 0 0 0 0 1	CHR$(&H81)

The method of designing patterns in each screen will vary depending on the number of colors available in each screen mode. This is because the number of bits per pixel is directly related to the numbers of colors available in each screen mode. In any screen, where X is the total number of colors available for that screen,

$$LOG_2(X) = Y$$

where Y is the number of bits per pixel. In high resolution, each byte of the string is able to plot eight points across the screen (one bit per pixel) since $LOG_2(2) = 1$.

However, in Screen 5, one medium resolution tile byte describes only two pixels since medium resolution has only two bits per pixel since $LOG_2(16) = 4$ bits per pixel. Every four bits of the tile byte describes one of 16 possible colors associated with each of the two pixels to be plotted.

The following chart shows the binary and hexadecimal values associated with the given colors in Screen 1.

Color Palette 0	**Color no. in binary**	**Pattern to draw solid line in binary**	**Pattern to draw solid line in hexadecimal**
green	01	01010101	&H55
red	10	10101010	&HAA
brown	11	11111111	&HFF

Color Palette 1	Color no. in binary	Pattern to draw solid line in binary	Pattern to draw solid line in hexadecimal
cyan	01	01010101	&H55
magenta	10	10101010	&HAA
white	11	11111111	&HFF

In medium resolution, SCREEN 1, the following example plots a pattern of boxes with a border color of red in palette 0 and magenta in palette 1.

```
PAINT (320,100),CHR$(&HAA)+CHR$(&H82)+CHR$(&H82)
+CHR$(&H82)+CHR$(&H82)+CHR$(&H82)+CHR$(&H82)
+CHR$(&HAA)
```

```
                 x increases -->
                 bit of Tile byte
                 7 6 5 4 3 2 1 0
Tile byte 0      1 0 1 0 1 0 1 0     CHR$(&HAA)
Tile byte 1      1 0 0 0 0 0 1 0     CHR$(&H82)
Tile byte 2      1 0 0 0 0 0 1 0     CHR$(&H82)
Tile byte 3      1 0 0 0 0 0 1 0     CHR$(&H82)
Tile byte 4      1 0 0 0 0 0 1 0     CHR$(&H82)
Tile byte 5      1 0 0 0 0 0 1 0     CHR$(&H82)
Tile byte 6      1 0 0 0 0 0 1 0     CHR$(&H82)
Tile byte 7      1 0 1 0 1 0 1 0     CHR$(&HAA)
```

Examples: The program below demonstrates how to tile an area with three lines of red, two lines of green, and one line of magenta.

```
10 CLS:SCREEN 3:KEY OFF
20 TIL$=CHR$(&H44)+CHR$(&H44)+CHR$(&H44)
   +CHR$(&H22)+CHR$(&H22)+CHR$(&H55)
40 VIEW (40,50)-(120,150),0,9
50 GOSUB 1000
70 GOTO 1020
1000 PAINT (80,100),TIL$,9
1010 RETURN
1020 END
```

PAINT Statement

The following example uses paint tiling with the *background* attribute.

```
10  SCREEN 1:CLS:COLOR 0,1
20  TIL$=CHR$(&H5F)+CHR$(&H5F)+CHR$(&H27)
        +CHR$(&H81)
30  VIEW (1,1)-(150,100),0,2
40  LOCATE 3,22:PRINT "<---Without back-"
50  LOCATE 4,22:PRINT "  ground tile"
60  PAINT (125,50),CHR$(&H5F)
70  PAINT (125,50),TIL$,2
80  '
90  'with background tile'
100 '
110 VIEW  (160,100)-(310,198),0,2
120 LOCATE 16,1:PRINT "With background-->
130 LOCATE 17,1:PRINT "tile chr$(&H5F)"
140 PAINT (125,50),CHR$(&H5F)
150 PAINT (125,50),TIL$,2,CHR$(&H5F)
160 LINE (1,100)-319,100),3
170 FOR I=1 TO 2500: NEXT I
180 END
```

PALETTE Statement

Purpose: Allows control of the hardware palette.

Versions:

Cassette	Cartridge	Compiler
	***	

Format: PALETTE [*attribute*][, *color*]

Remarks:

attribute is an expression which evaluates to an integer value in the range of 0 to 15 that indicates the attribute that is to be changed.

color is an expression that evaluates to an integer value which is a color in the range of 0 to 15 to be assigned to attribute. In low resolution, there are 16 attributes available (0 to 15). In medium resolution, there are 4 (0 to 3) or 16 (0 to 15) attributes available, depending on the current screen mode. In high resolution, there are 2 (0 to 1) or 4 (0 to 3) attributes available, depending on the current screen mode. The default attribute is always the maximum attribute for the current screen mode. For more information see "Graphics Modes" in Chapter 3.

The PALETTE statement without any arguments can be used to reset all palettes to their original color.

The PALETTE statement can also be used to change the color of text. PALETTE used in conjunction with the

PALETTE
Statement

COLOR statement allows as much flexibility with text in graphics mode as in text mode. PALETTE can be used to give the effect of making lines disappear. You can draw an object in a given color and then use the PALETTE statement to change the object to the background color making the object seem to disappear.

When you want to assign colors to many attributes quickly you should use the PALETTE USING statement. See the "PALETTE USING Statement" in this chapter for more information.

Example: The PALETTE statement allows you to change the color of any object or text on the screen. For example, the following statement draws a blue line.

```
LINE (0,0)-(100,100),1
```

The PALETTE statement can be used to change the color of the line.

```
PALETTE 1,3
```

This statement says that every time the system sees the attribute 1 you will see the color 3 (cyan). From this point on when you reference the color attribute 1, you will actually see the color cyan.

Purpose: Allows the setting of all palette entries with one statement.

Versions: Cassette Cartridge Compiler
***

Cassette	Cartridge	Compiler
	***	

Format: PALETTE USING [*arrayname*(*starting*)]

Remarks:

array name	is an integer array which must have 16 elements indexed from the starting index.
starting	is an integer indicating the starting position in the array for assigning the palette colors.

PALETTE USING allows you to quickly assign the colors for many attributes simultaneously. Any entry in the array with a value of -1 will not modify the current color for that attribute. PALETTE USING is valid for both text and graphic modes.

PALETTE USING
Statement

Example:

```
5   DEFINT A-Z        'define all variables integer
10  SCREEN 1:CLS:
20  LINE (50,50)-(120,120),1,BF
30  LINE (60,60)-(110,110),2,BF
40  LINE (70,70)-(100,100),3,BF
50  LINE (80,80)-(90,90),0,BF
60  DIM PAL (16)
70  DATA 14,4,5,6,-1,-1,-1,-1
75  DATA -1,-1,-1,-1,-1,-1,-1,-1
80  FOR I = 0 TO 15
90  READ PAL(I)
100 NEXT I
110 FOR T=1 TO 1000: NEXT T     'delay loop
200 PALETTE USING PAL(0)        'change colors
300 END
```

The attributes of 0, 1, 2, and 3 have been reassigned colors 14, 4, 5, and 6, respectively. The rest of the colors remain unchanged. Statement 200 changes the colors on the screen from black, blue, green, and cyan to yellow, red, magenta, and brown. Attribute 1 now corresponds to color 14; attribute 2 now corresponds to red, and so on.

Attribute	Old Color	New Color
0	Black	14-Yellow
1	Blue	4-Red
2	Green	5-Magenta
3	Cyan	6-Brown'
4	Red	-1-Unchanged
5	Magenta	-1-Unchanged
6	Brown	-1-Unchanged
7	White	-1-Unchanged
8	Gray	-1-Unchanged
9	Light Blue	-1-Unchanged
10	Light Green	-1-Unchanged
11	Light Cyan	-1-Unchanged
12	Light Red	-1-Unchanged

Attribute	Old Color	New Color
13	Light Magenta	-1-Unchanged
14	Yellow	-1-Unchanged
15	Hi-Int-White	-1-Unchanged

PCOPY Statement

Purpose: Allows copying from one screen page to another in all screen modes.

Versions:

Cassette	Cartridge	Compiler
	***	

Format: PCOPY [source page] , [destination page]

Remarks:

source page is an integer expression in the range of the number of pages which is determined by the current video memory size and the size per page for the current screen mode.

destination page
The destination page has the same requirements as the source page. For more information on pages and memory allocation, refer to the CLEAR statement, the SCREEN statement, and the section in Chapter 3 on graphics.

Example:

```
PCOPY 1,2
```

This copies the contents of page 1 to page 2.

Purpose: Returns the byte read from the indicated memory position.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: *v* = PEEK(*n*)

Remarks:

n is an integer in the range 0 to 65535. *n* is the offset from the current segment as defined by the DEF SEG statement, and indicates the address of the memory location to be read. (See "DEF SEG Statement" in this chapter.)

The returned value will be an integer in the range 0 to 255.

PEEK is the complementary function to the POKE statement (see "POKE Statement," later in this chapter).

PEN
Statement and Function

Purpose: Reads the light pen.

Versions:

Cassette	Cartridge	Compiler
***	***	(**)

PEN STOP only in Cartridge BASIC and the BASIC Compiler.

Format: As a statement:

PEN ON

PEN OFF

PEN STOP

As a function:

v = PEN(*n*)

Remarks: The PEN function, *v*=PEN(*n*), reads the light pen coordinates.

n is a numeric expression in the range 0 to 9, and affects the value returned by the function as follows:

0 A flag indicating if pen was down since last poll. Returns -1 if down, 0 if not.

1 Returns the x coordinate where pen was last activated. The range is 0 to 159 in low resolution, 0 to 319 in medium resolution, and 0 to 639 in high resolution.

2	Returns the y coordinate where pen was last activated. The range is 0 to 199.
3	Returns the current pen switch value. -1 if down, 0 if up.
4	Returns the last known valid x coordinate. The range is 0 to 159 in low resolution, 0 to 319 in medium resolution, and 0 to 639 in high resolution.
5	Returns the last known valid y coordinate. The range is 0 to 199.
6	Returns the character row position where pen was last activated. The range is 1 to 24.
7	Returns the character column position where pen was last activated. The range is 1 to 20, 1 to 40 or 1 to 80 depending on WIDTH.
8	Returns the last known valid character row. The range is 1 to 24.
9	Returns the last known valid character column position. The range is 1 to 20, 1 to 40 or 1 to 80 depending on WIDTH.

PEN ON enables the PEN read function. The PEN function is initially off. A PEN ON statement must be executed before any pen read function calls can be made. A call to the PEN function while the PEN function is off results in an **Illegal function call** error.

PEN
Statement and Function

Conversely, to improve execution speed, it is a good idea to turn the pen off with a PEN OFF statement when you are not using the light pen.

For Cartridge BASIC, executing PEN ON will also allow trapping to take place with the ON PEN statement. After PEN ON, if a nonzero line number was specified in the ON PEN statement, then every time the program starts a new statement BASIC checks to see if the pen was activated. Refer to "ON PEN Statement" in this chapter.

PEN OFF disables the PEN read function. For Cartridge BASIC, no trapping of the pen takes place and action by the light pen is not remembered even if it does take place.

PEN STOP is only available in Cartridge BASIC. It disables trapping of light pen activity, but if activity happens it is remembered so an immediate trap occurs when a PEN ON is executed.

When the pen is down in the border area of the screen, the values returned are inaccurate.

You should not try I/O to cassette while PEN is ON.

Example:

```
50 PEN ON
60 FOR I=1 TO 500
70 X=PEN(0): X1=PEN(3)
80 PRINT X, X1
90 NEXT
100 PEN OFF
```

This example prints the pen value since the last poll, and the current value.

Purpose: Plays music as specified by *string*.

Versions: Cassette Cartridge Compiler
*** (**)

Format: PLAY *string* [,[*string*] [, *string*]]

Remarks: PLAY is a statement similar to DRAW because it imbeds a "tune definition language" into a character string.

string is a string expression consisting of single character music commands. In Cartridge BASIC, PLAY supports three separate strings to allow independent control of each of three voices (only when SOUND is ON).

The single character commands in PLAY are:

A to G with optional #, +, or -
Plays the indicated note in the current octave. A number sign (#) or plus sign (+) afterwards indicates a sharp, a minus sign (-) indicates a flat. The #, +, or - is not allowed unless it corresponds to a black key on a piano. For example, B# is an invalid note.

O n Octave. Sets the current octave for the following notes. There are 7 octaves, numbered 0 to 6. Each octave goes from C to B. Octave 3 starts with middle C. Octave 4 is the default octave.

The lowest frequency the multi-voice sound chip can produce is 110 Hz. This

corresponds to the note A of octave 0. If you try to play a note below 110 Hz BASIC will not give an error, but will play the note A for all notes below 110 Hz. See the "SOUND Statement" in this chapter for information on notes and their corresponding frequencies.

> n Go up to the next higher octave and play note n. Each time note n is played, the octave goes up, until it reaches octave 6. For example, PLAY ">A" raises the octave and plays note A. Each time PLAY ">A" is executed, the octave goes up until it reaches octave 6; then each time PLAY ">A" executes, note A plays at octave 6. Cartridge BASIC only.

< n Go down one octave and play note n. Each time note n is played, the octave goes down, until it reaches octave 0. For example, PLAY "<A" lowers the octave and plays note A. Each time PLAY "<A" is executed, the octave goes down until it reaches octave 0, then each time PLAY "<A" executes, note A plays at octave 0. Cartridge BASIC only.

N n Plays note n. n may range from 0 to 84. In the 7 possible octaves, there are 84 notes. n=0 means rest. This is an alternative way of selecting notes besides specifying the octave (O n) and the note name (A-G).

L n Sets the length of the following notes. The actual note length is 1/n. n may range from 1 to 64. The following table may help explain this:

Length	Equivalent
L1	whole note
L2	half note
L3	one of a triplet of three half notes (1/3 of a 4 beat measure)
L4	quarter note
L5	one of a quintuplet (1/5 of a measure)
L6	one of a quarter note triplet
	.
	.
	.
L64	sixty-fourth note

The length may also follow the note when you want to change the length only for the note. For example, A16 is equivalent to L16A.

P n Pause (rest). n may range from 1 to 64, and figures the length of the pause in the same way as L (length).

. (dot or period) After a note, causes the note to be played as a dotted note. That is, its length is multiplied by 3/2. More than one dot may appear after the note, and the length is adjusted accordingly. For example, "A.." will play 9/4 as long as L specifies, "A..." will play 27/8 as long, etc. Dots may also appear after a pause (P) to scale the pause length in the same way.

T n Tempo. Sets the number of quarter notes in a minute. n may range from 32 to 255. The default is 120. Under "SOUND

Statement," later in this chapter, is a table listing common tempos and the equivalent beats per minute.

MF Music foreground. Music (created by SOUND or PLAY) runs in foreground. That is, each subsequent note or sound will not start until the previous note or sound is finished. You can press the Fn key followed the Break key to exit PLAY.

MB Music background. Music (created by SOUND or PLAY) runs in background instead of in foreground. That is, each note or sound is placed in a buffer allowing the BASIC program to continue executing while music plays in the background. Up to 32 notes (or rests) may be played in background at a time. Music background is the default state.

MN Music normal. Each note plays 7/8 of the time specified by L (length). This is the default setting of **MN, ML,** and **MS**.

ML Music legato. Each note plays the full period set by L (length).

MS Music staccato. Each note plays 3/4 of the time specified by L.

X variable;
Executes specified string.

V Volume. Valid only when SOUND is turned ON. Sets volume in the range 0 to 15. The default value is 8.

In all these commands the *n* argument can be a constant like **12** or it can be =***variable***; where *variable* is the name of a variable. The semicolon (;) is required when you use a variable in this way, and when you use the **X** command. Otherwise, a semicolon is optional between commands. A semicolon is not allowed after MF, MB, MN, ML, or MS. Also, any blanks in *string* are ignored.

You can also specify variables in the form VARPTR$(*variable*), instead of =*variable*;. The VARPTR$ form is the only one that can be used in compiled programs. For example:

One Method	**Alternative Method**
PLAY "XA$;"	PLAY "X"+VARPTR$(A$)
PLAY "O=I;"	PLAY "O="+VARPTR$(I)

You can use **X** to store a "subtune" in one string and call it over again with different tempos or octaves from another string.

Examples: The following example plays a tune.

```
10 REM little lamb
20 MARY$="GFE-FGGG"
30 PLAY "MB T100 O3 L8;XMARY$;P8 FFF4"
40 PLAY "GB-B-4; XMARY$; GFFGFE-."
```

The following example plays the scale from octave 0 to octave 6.

```
10 ' Play the scale using > octave
20 SCALE$="CDEFGAB"
30 PLAY "O0 XSCALE$;"
40 FOR I=1 TO 6
50 PLAY ">XSCALE$;"
60 NEXT
70 ' Play the scale using < octave
80 PLAY "O6 XSCALE$;"
90 FOR I=1 TO 6
100 PLAY "<XSCALE$;"
110 NEXT
```

The following example shows the use of multiple voices.

```
10 SOUND ON
20 CLS
30 PRINT "Turn the external speaker on for demo"
50 PLAY "MBMLO1T255","O2MLT255","O3MLT255"
100 A$="CDECCDEC"
110 B$="EFGGEFGG"
120 C$="CGCCCGCC"
140 FOR L=32 TO 4 STEP -4
145 PRINT "Turn the external speaker on for demo"
150 PLAY "1=1;","1=1;","1=1;"
200 PLAY A$,B$,C$
210 PLAY B$,C$,A$
220 PLAY C$,A$,B$
225 NEXT L
230 GOTO 140
```

PLAY(n) Function

Purpose: Returns the number of notes currently in the music background buffer.

Versions:

Cassette	Cartridge	Compiler
	***	

Format: *v*=PLAY(*n*)

Remarks:

n is a dummy argument that can have any value when SOUND is turned OFF.

When SOUND is turned ON n may have one of the following values:

0 returns the number of notes left to play on voice channel 0

1 returns the number of notes left to play on voice channel 1

2 returns the number of notes left to play on voice channel 2

If n is not 0, 1, or 2, then the notes remaining on voice channel 0 will be returned.

The maximum value that can be returned is 32, the maximum number of notes held in the buffer.

PLAY(*n*) returns a 0 when the program is running in music foreground mode.

Example:

PLAY(n)
Function

```
10 'when 5 notes are left in the music buffer
20 'go to line 1000 and play another tune
30 PLAY "MB CDEFGAB"
40 IF PLAY(0)= 5 GOTO 1000
.
.
.
1000 PLAY "MB 04 T200 L4 MS GG#GE"
```

Purpose: Maps physical coordinates to world coordinates or world coordinates to physical coordinates.

Versions:

Cassette	Cartridge	Compiler
	***	

Graphics mode only.

Format: *v*=PMAP(*x*,*n*)

Remarks:

x coordinate of the point that is to be mapped

n may be a value in the range 0 to 3 such that

- **0** maps the world coordinate x to the physical coordinate x
- **1** maps the world coordinate y to the physical coordinate y
- **2** maps the physical coordinate x to the world coordinate x
- **3** maps the physical coordinate y to the world coordinate y

PMAP is used to translate coordinates between the world system as defined by the WINDOW statement and the physical coordinate system as defined by the VIEW statement.

PMAP(x,0) and PMAP(x,1) are used to map values from the world coordinate system to the physical coordinate system.

STATEMENTS

PMAP
Function

PMAP(x,2) and PMAP(x,3) are used to map values from the physical coordinate system to the world coordinate system.

For example, if the statement

```
SCREEN 1: WINDOW (-1,-1)-(1,1)
```

is in effect we can use PMAP to map the world coordinate points of (-1,-1) and (1,1) to their corresponding physical points on the screen.

PMAP(-1,0) returns the physical x coordinate value of 0.

PMAP(-1,1) returns the physical y coordinate value of 199.

PMAP(1,0) returns the physical x coordinate value of 319.

PMAP(1,1) returns the physical y coordinate value of 0.

The above information tells us that the point (-1,-1) which is in the lower left corner of the screen corresponds to the physical point (0,199). We also know that the point (1,1) which is in the upper right corner corresponds to the physical point (319,0).

Purpose: Returns the color of the specified point on the screen or current graphics coordinate or returns the value of the current x or y coordinate.

Versions:

Cassette	Cartridge	Compiler
***	***	(**)

Graphics mode only.

Format: *v* = POINT (*x,y*)

v = POINT (*n*)

Remarks:

(x,y) are the coordinates of the point to be used. The coordinates must be in absolute form (see "Specifying Coordinates"under "Graphics Modes" in Chapter 3).

If the point given is out of range the value -1 is returned. In low resolution, valid returns are 0 to 15. In medium resolution, valid returns are 0 to 3 or 0 to 15, depending on the screen mode. In high resolution, valid returns are 0 to 1 or 0 to 3, depending on the current screen mode. For more information see "Graphics Modes" in Chapter 3.

n returns the value of the current x or y graphics coordinate. *n* can have a value from 0 to 3 where:

0 returns the current physical x coordinate.

1 returns the current physical y coordinate.

2 returns the current world x coordinate if WINDOW is active. If WINDOW is not active, returns the current physical x coordinate.

3 returns the current world y coordinate if WINDOW is active. If WINDOW is not active, returns the current physical y coordinate.

For more information, see "WINDOW Statement" in this chapter.

Examples: The following example inverts the current setting of point (I,I).

```
5 SCREEN 2
10 IF POINT(I,I)<>0 THEN PRESET(I,I)
                     ELSE PSET(I,I)
```

or

```
10 PSET(I,I),1-POINT(I,I)
```

The following example illustrates values returned by the POINT function. Note the change in the values depending upon WINDOW.

```
10 CLS: SCREEN 1,0
15 PRINT "POINT(n) with WINDOW inactive"
20 GOSUB 100
30 WINDOW (0,0)-(319,199)
40 PRINT "POINT(n) with WINDOW active"
50 GOSUB 100
60 PRINT "POINT(n), WINDOW and SCREEN active"
70 WINDOW SCREEN (0,0)-(319,199)
80 GOSUB 100
90 END
100 PSET (5,15)
110 FOR I=0 TO 3
120 PRINT POINT (I);
130 NEXT
135 PRINT: PRINT
140 RETURN
Ok
RUN

POINT(n) with WINDOW inactive
5  15  5  15
POINT(n) with WINDOW active
5  184  5  15
POINT(n), WINDOW and SCREEN active
5  15  5  15
```

POKE Statement

Purpose: Writes a byte into a memory location.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: POKE *n,m*

Remarks:

n must be in the range 0 to 65535 and indicates the address of the memory location where the data is to be written. It is an offset from the current segment as defined by the DEF SEG statement (see "DEF SEG Statement" in this chapter).

m m is the data to be written to the specified location. It must be in the range 0 to 255.

The complementary function to POKE is PEEK. (See "PEEK Function" in this chapter.) POKE and PEEK are useful for efficient data storage, loading machine language subroutines, and passing arguments and results to and from machine language subroutines.

> **Warning:** BASIC does not do any checking on the address. So don't go POKEing around in BASIC's stack, BASIC's variable area, or your BASIC program.

Purpose: Returns the current cursor column position.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: *v* = POS(*n*)

Remarks: *n* is a dummy argument.

The current horizontal (column) position of the cursor is returned. The returned value will be in the range 1 to 20, 1 to 40 or 1 to 80, depending on the current WIDTH setting. CSRLIN can be used to find the vertical (row) position of the cursor (see "CSRLIN Variable" in this chapter).

Also see the LPOS function.

Example:

```
IF POS(0)>60 THEN PRINT CHR$(13)
```

This example prints a carriage return (moves the cursor to the beginning of the next line) if the cursor is beyond position 60 on the screen.

PRINT
Statement

Purpose: Displays data on the screen.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: PRINT [*list of expressions*] [;]

? [*list of expressions*] [;]

Remarks:

list of expressions is a list of numeric and/or string expressions, separated by commas, blanks, or semicolons. Any string constants in the list must be enclosed in quotation marks.

If the list of expressions is omitted, a blank line is displayed. If the list of expressions is included, the values of the expressions are displayed on the screen.

> **Note:** The question mark (?) may be used as a short way of entering PRINT only when you are using the BASIC program editor.

Print Positions

The position of each printed item depends on the punctuation used to separate the items in the list. BASIC divides the line into print zones of 14 spaces each.

In the list of expressions:

- Typing a comma causes the next value to be printed at the beginning of the next zone.

- Typing a semicolon causes the next value to be printed immediately after the last value.

- Typing one or more spaces between expressions has the same effect as typing a semicolon.

If a comma, semicolon, or SPC or TAB function ends the list of expressions, the next PRINT statement begins printing on the same line, spacing accordingly. If the list of expressions ends without a comma, semicolon, SPC or TAB function, a carriage return is printed at the end of the line; that is, BASIC moves the cursor to the beginning of the next line.

If the value to be printed is longer than the number of character positions on the current line, then the value will be printed at the beginning of the next line. If the value to be printed is longer than the defined WIDTH, BASIC prints as much as it can on the current line and prints the rest on the next line.

Scrolling occurs as described under "Text Mode" in Chapter 3.

Printed numbers are always followed by a space. Positive numbers are preceded by a space. Negative numbers are preceded by a minus sign.
Single-precision numbers that can be represented with 7 or fewer digits in fixed point format as accurately as they can be represented in the floating point format, are output using fixed point or integer format. For example, 10^(-7) is output as .0000001 and 10^(-8) is output as 1E-8.

PRINT
Statement

BASIC automatically places a carriage return/line feed after printing *width* characters, where *width* is 20, 40 or 80, as defined by the WIDTH statement. This causes two lines to be skipped when you print exactly 20, 40 or 80 characters, unless the PRINT statement ends in a semicolon (;).

LPRINT is used to print information on the printer. See "LPRINT and LPRINT USING Statements" earlier in this chapter.

Example:

```
Ok
10 X=5
20 PRINT X+5, X-5, X*(-5)
30 END
RUN
 10            0             -25
Ok
```

In this example, the commas in the PRINT statement cause each value to be printed at the beginning of the next print zone.

```
Ok
10 INPUT X
20 PRINT X "SQUARED IS" X^2 "AND";
30 PRINT X "CUBED IS" X^3
RUN
? 9
 9 SQUARED IS 81 AND 9 CUBED IS 729
Ok
RUN
? 21
 21 SQUARED IS 441 AND 21 CUBED IS 9261
Ok
```

Here, the semicolon at the end of line 20 causes both PRINT statements to be printed on the same line.

```
Ok
10 FOR X = 1 TO 5
20 J=J+5
30 K=K+10
40 ?J;K;
50 NEXT X
RUN
 5  10  10  20  15  30  20  40  25  50
Ok
```

Here, the semicolons in the PRINT statement cause each value to be printed immediately after the preceding value. (Don't forget, a number is always followed by a space and positive numbers are preceded by a space.) In line 40, a question mark is used instead of the word PRINT.

PRINT USING Statement

Purpose: Prints strings or numbers using a specified format.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: PRINT USING *v$; list of expressions* [;]

Remarks:

v$ is a string constant or variable which consists of special formatting characters. These formatting characters (see below) determine the field and the format of the printed strings or numbers.

list of expressions
consists of the string expressions or numeric expressions that are to be printed, separated by semicolons or commas.

String Fields

When PRINT USING is used to print strings, one of three formatting characters may be used to format the string field:

! Specifies that only the first character in the given string is to be printed.

\n spaces Specifies that 2+n characters from the string are to be printed. If the backslashes are typed with no spaces, two characters are printed; with one space, three characters are printed, and

so on. If the string is longer than the field, the extra characters are ignored. If the field is longer than the string, the string is left-justified in the field and padded with spaces on the right.

Example:

```
10 A$="LOOK": B$="OUT"
30 PRINT USING "!";A$;B$
40 PRINT USING "\  \";A$;B$
50 PRINT USING "\   \";A$;B$;"!!"
RUN
LO
LOOKOUT
LOOK OUT  !!
```

& Specifies a variable length string field. When the field is specified with "&," the string is output exactly as input. Example:

```
10 A$="LOOK": B$="OUT"
20 PRINT USING "!";A$;
30 PRINT USING "&";B$
RUN
LOUT
```


Numeric Fields

When PRINT USING is used to print numbers, the following special characters may be used to format the numeric field:

A number sign is used to represent each digit position. Digit positions are always filled. If the number to be printed has

fewer digits than positions specified, the number is right-justified (preceded by spaces) in the field.

. A decimal point may be inserted at any position in the field. If the format string specifies that a digit is to precede the decimal point, the digit will always be printed (as 0 if necessary). Numbers are rounded as necessary.

```
PRINT USING "##.##";.78
 0.78

PRINT USING "###.##";987.654
987.65

PRINT USING "##.##   ";10.2,5.3,66.789,.2
10.20    5.30   66.79    0.23
```

In the last example, three spaces were inserted at the end of the format string to separate the printed values on the line.

\+ A plus sign at the beginning or end of the format string causes the sign of the number (plus or minus) to be printed before or after the number.

\- A minus sign at the end of the format field causes negative numbers to be printed with a trailing minus sign.

```
PRINT USING "+##.##   ";-68.95,2.4,55.6,
-68.95    +2.40   +55.60    -0.90

PRINT USING "##.##-   ";-68.95,22.449,-7
68.95-   22.45     7.01-
```

** A double asterisk at the beginning of the format string causes leading spaces

in the numeric field to be filled with asterisks. The ** also specifies positions for two more digits.

```
PRINT USING "**#.#   ";12.39,-0.9,765.1
*12.4   *-0.9   765.1
```

$$ A double dollar sign causes a dollar sign to be printed to the immediate left of the formatted number. The $$ specifies two more digit positions, one of which is the dollar sign. The exponential format cannot be used with $$. Negative numbers cannot be used unless the minus sign trails to the right.

```
PRINT USING "$$###.##";456.78
 $456.78
```

****$** The **$ at the beginning of a format string combines the effects of the above two symbols. Leading spaces are filled with asterisks and a dollar sign will be printed before the number. **$ specifies three more digit positions, one of which is the dollar sign.

```
PRINT USING "**$##.##";2.34
***$2.34
```

, A comma that is to the left of the decimal point in a formatting string causes a comma to be printed to the left of every third digit to the left of the decimal point. A comma that is at the end of the format string is printed as part of the string. A comma specifies another digit position. The comma has no effect if used with the exponential (^^^^) format.

PRINT USING
Statement

```
PRINT USING "####,.##";1234.5
1,234.50
```

```
PRINT USING "####.##,";1234.5
1234.50,
```

^^^^ Four carets may be placed after the digit position characters to specify exponential format. The four carets allow space for E+nn or D+nn to be printed. Any decimal point position may be specified. The significant digits are left-justified, and the exponent is adjusted. Unless a leading + or trailing + or - is specified, one digit position is used to the left of the decimal point to print a space or a minus sign.

```
Ok
PRINT USING "##.##^^^^";234.56
 2.35E+02
Ok
PRINT USING ".###^^^^-";-88888
.889E+05-
Ok
PRINT USING "+.##^^^^";123
+.12E+03
Ok
```

_ An underscore in the format string causes the next character to be output as a literal character.

```
PRINT USING "_!##.##_!";12.34
!12.34!
```

The literal character itself may be an underscore by placing two underscores "__" in the format string.

If the number to be printed is larger than the specified numeric field, a percent sign (%) is printed

in front of the number. If rounding causes the number to exceed the field, the percent sign is printed in front of the rounded number.

```
Ok
PRINT USING "##.##";111.22
%111.22
Ok
PRINT USING ".##";.999
%1.00
Ok
```

If the number of digits specified exceeds 24, an **Illegal function call** error occurs.

Example: This example shows how you can include string constants in the format string.

```
Ok
PRINT USING "THIS IS EXAMPLE _##"; 1
THIS IS EXAMPLE #1
Ok
```

PRINT # and PRINT # USING Statements

Purpose: Writes data sequentially to a file.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: PRINT #*filenum*, [USING *x$*;] *list of exps* [;]

Remarks:

filenum is the number used when the file was opened for output.

x$ is a string expression comprised of formatting characters as described in the previous section, "PRINT USING Statement."

list of exps is a list of the numeric and/or string expressions that will be written to the file.

PRINT # does not compress data on the file. An image of the data is written to the file just as it would be displayed on the screen with a PRINT statement. For this reason, care should be taken to delimit the data on the file, so that it will be input correctly from the file.

In the list of expressions, numeric expressions should be delimited by semicolons. For example,

```
PRINT #1,A;B;C;X;Y;Z
```

(If commas are used as delimiters, the extra blanks that are inserted between print fields are also written to the file.)

String expressions must be separated by semicolons in the list. To format the string expressions correctly on the file, use explicit delimiters in the list of expressions.

For example, let A$="CAMERA" and B$="93604-1." The statement

```
PRINT #1,A$;B$
```

writes CAMERA93604-1 to the file. Because there are no delimiters, this could not be input as two separate strings. To correct the problem, insert explicit delimiters into the PRINT # statement as follows:

```
PRINT #1,A$;",";B$
```

The image written to the file is

```
CAMERA,93604-1
```

which can be read back into two string variables.

If the strings themselves contain commas, semicolons, significant leading blanks, carriage returns, or line feeds, write them to the file surrounded by explicit quotation marks using CHR$(34).

For example, let A$="CAMERA, AUTOMATIC" and B$="   93604-1." The statement:

```
PRINT #1,A$;B$
```

writes the following image to the file:

```
CAMERA, AUTOMATIC   93604-1
```

and the statement:

PRINT # and PRINT # USING Statements

```
INPUT #1,A$,B$
```

inputs the string "CAMERA" to A$ and
"AUTOMATIC   93604-1" to B$.

To separate these strings properly on the file, write double quotes to the file image using CHR$(34). The statement:

```
PRINT #1,CHR$(34);A$;CHR$(34);CHR$(34);B$;CHR$(34)
```

writes the following image to the file:

```
"CAMERA, AUTOMATIC""   93604-1"
```

and the statement:

```
INPUT #1,A$,B$
```

inputs "CAMERA, AUTOMATIC" to A$ and
"   93604-1" to B$.

The PRINT # statement may also be used with the USING option to control the format of the file. For example:

```
PRINT #1,USING"$$###.##,";J;K;L
```

The easy way to avoid all these problems is to use the WRITE # statement rather than the PRINT # statement. (Refer to "WRITE # Statement," at the end of this chapter.)

Example: For more examples using PRINT # and WRITE #, see Appendix B, "BASIC Diskette Input and Output."

PSET and PRESET Statements

Purpose: Draws a point at the specified position on the screen.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Graphics mode only.

Format: PSET (*x,y*) [*,attribute*]

PRESET (*x,y*) [*,attribute*]

Remarks:

(x,y) are the coordinates of the point to be set. They may be in absolute or relative form, as explained in the section "Specifying Coordinates"under "Graphics Modes" in Chapter 3.

attribute is an integer or integer expression in the range of 0 to 15. In low resolution, there are 16 attributes available (0 to 15). In medium resolution, there are 4 (0 to 3) or 16 (0 to 15) attributes available, depending on the current screen mode. In high resolution, there are 2 (0 to 1) or 4 (0 to 3) attributes available, depending on the current screen mode. The default attribute is always the maximum attribute. 0 is the background attribute. For more information see "Graphics Mode" in Chapter 3.

Screen modes 3-6 are not supported in the BASIC Compiler.

PSET and PRESET Statements

PRESET is almost identical to PSET. The only difference is that if no *attribute* parameter is given to PRESET, the background attribute (0) is selected. If *attribute* is included, PRESET is identical to PSET. Line 70 in the example below could just as easily be:

```
70 PSET(I,I),0
```

If an out of range coordinate is given to PSET or PRESET no action is taken nor is an error given. If *attribute* is greater than 15, an **Illegal function call** error results.

Example: Lines 20 through 40 of this example draw a diagonal line from the point (0,0) to the point (100,100). Then lines 60 through 80 erase the line by setting each point to a color of 0.

```
10 SCREEN 1
20 FOR I=0 TO 100
30 PSET (I,I)
40 NEXT
50 'erase line
60 FOR I=100 TO 0 STEP -1
70 PRESET(I,I)
80 NEXT
```

Purpose: Writes a record from a random buffer to a random file.

> **Note:** This statement requires the use of DOS 2.10.

Versions:

Cassette	Cartridge	Compiler
	***	***

Format: PUT [#]*filenum* [, *number*]

Remarks:

filenum is the number under which the file was opened.

number is the record number for the record to be written, in the range of 1 to 16,777,215. In the BASIC Compiler, *number* must be in the range of 1 to 32767.

If *number* is omitted, the record has the next available record number (after the last PUT).

PRINT #, PRINT # USING, WRITE #, LSET, and RSET may be used to put characters in the random file buffer before a PUT statement. In the case of WRITE #, BASIC pads the buffer with spaces up to the carriage return.

Any attempt to read or write past the end of the buffer causes a **Field overflow** error. Refer to Appendix B, "BASIC Diskette Input and Output."

Because BASIC and DOS block as many records as possible in 512 byte sectors, the PUT statement does not necessarily perform a physical write to the diskette.

PUT can be used for a communications file. In that case *number* is the number of bytes to write to the communications file. This number must be less than or equal to the value set by the LEN option on the OPEN "COM... statement.

Example: See Appendix B.

Purpose: Writes colors onto a specified area of the screen.

Versions:

Cassette	Cartridge	Compiler
	***	***

Graphics mode only.

Format: PUT *(x,y)* *,array* [*,action*]

Remarks:

(x,y) are the coordinates of the top left corner of the image to be transferred.

array is the name of a numeric array containing the information to be transferred. See "GET Statement (Graphics)" in this chapter for more information on this array.

action is one of:

PSET
PRESET
XOR
OR
AND

XOR is the default.

PUT is the opposite of GET in the sense that it takes data out of the array and puts it onto the screen. However it also provides the option of interacting with the data already on the screen using the action.

PSET as an action simply stores the data from the array onto the screen, so this is the true opposite of GET.

PRESET is the same as PSET except a negative image is produced. That is, a value of 0 in the array causes the corresponding point to have attribute number 3, and vice versa; a value of 1 in the array causes the corresponding point to have attribute 2, and vice versa.

AND is used when you want to transfer the image only if an image already exists under the transferred image.

OR is used to superimpose the image onto the existing image.

XOR is a special mode which may be used for animation. XOR causes the points on the screen to be inverted where a point exists in the array image. XOR has a unique property that makes it especially useful for animation: when an image is PUT against a complex background *twice*, the background is restored unchanged. This allows you to move an object around without obliterating the background.

In medium resolution mode (SCREEN 1 or SCREEN 4), the AND, XOR, and OR operations have the following effects on attribute selections:

AND

screen	array value 0	1	2	3
0	0	0	0	0
1	0	1	0	1
2	0	0	2	2
3	0	1	2	3

OR

screen	array value 0	1	2	3
0	0	1	2	3
1	1	1	3	3
2	2	3	2	3
3	3	3	3	3

XOR

screen	array value 0	1	2	3
0	0	1	2	3
1	1	0	3	2
2	2	3	0	1
3	3	2	1	0

Animation of an object can be performed as follows:

1. PUT the object on the screen (with XOR)

2. Recalculate the new position of the object

3. PUT the object on the screen (with XOR) a second time at the old location to remove the old image.

4. Go to step 1, this time putting the object at the new location.

Movement done this way leaves the background unchanged. Flicker can be reduced by minimizing the time between steps 4 and 1, and making sure there is enough time delay between steps 1 and 3. If more than one object is being animated, every object should be processed at once, one step at a time.

If it is not important to preserve the background, animation can be performed using the PSET action verb. But you should remember to have an image area that will contain the "before" and "after" images of the object. This way the extra area will effectively erase the old image. This method may be somewhat faster than the method using XOR

described above, since only one PUT is required to move an object (although you must PUT a larger image).

If the image to be transferred is too large to fit on the screen, an **Illegal function call** error occurs.

RANDOMIZE Statement

Purpose: Reseeds the random number generator.

Versions:

Cassette	Cartridge	Compiler
***	***	(**)

Format: RANDOMIZE [*n*]

RANDOMIZE TIMER

Remarks:

n is an integer, single- or double-precision expression that is used as the random number seed. In Cassette BASIC, *n* must be an integer expression.

If *n* is omitted, BASIC suspends program execution and asks for a value by displaying:

```
Random Number Seed (-32768 to 32767)?
```

before executing RANDOMIZE.

If the random number generator is not reseeded, the RND function returns the same sequence of random numbers each time the program is run. To change the sequence of random numbers every time the program is run, place a RANDOMIZE statement at the beginning of the program and change the seed with each run.

In Cartridge BASIC, if DOS is present, the internal clock can be a useful way to get a random number seed. You can use VAL to change the last two digits of TIME$ to a number, and use that number for *n*.

In Cartridge BASIC, with DOS present, you can get a new random number seed without being prompted. To do this, use the TIMER function in the expression. Note that the sequence is different each time the programs runs.

Example:

```
10 RANDOMIZE
20 FOR I=1 TO 4
30 PRINT RND;
40 NEXT I
RUN
Random Number Seed (-32768 to 32767)?
```

Suppose you respond with 3. The program continues:

```
Random Number Seed (-32768 to 32767)? 3
 .7655695  .3558607  .3742327  .1388798
Ok
RUN
Random Number Seed (-32768 to 32767)?
```

Suppose this time you respond with 4. The program continues:

```
Random Number Seed (-32768 to 32767)? 4
 .1719568  .5273236  .6879686  .713297
Ok
RUN
Random Number Seed (-32768 to 32767)?
```

If you try 3 again, you'll get the same sequence as the first run:

```
Random Number Seed (-32768 to 32767)? 3
 .7655695  .3558607  .3742327  .1388798
Ok
```

In the program below, note that each time the program is run you see a different sequence of numbers.

RANDOMIZE
Statement

```
10 RANDOMIZE TIMER
20 FOR I=1 TO 4
30 PRINT RND;
40 NEXT
RUN
 .9590051   .1036786   .1464037   .7754918
Ok
RUN
 .8261163   .17422   .9191545   .5041142
Ok
```

Purpose: Reads values from a DATA statement and assigns them to variables (see "DATA Statement" in this chapter).

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: READ *variable* [,*variable*]...

Remarks:

variable is a numeric or string variable or array element which is to receive the value read from the DATA table.

A READ statement must always be used with a DATA statement. READ statements assign DATA statement values to the variables in the READ statement on a one-to-one basis. READ statement variables may be numeric or string, and the values read must agree with the variable types specified. If they do not agree, a **Syntax error** results.

A single READ statement may access one or more DATA statements (they will be accessed in order), or several READ statements may access the same DATA statement. If the number of variables in the list of variables exceeds the number of elements in the DATA statement(s), an **Out of data** error occurs. If the number of variables specified is fewer than the number of elements in the DATA statement(s), the READ statements following it begin reading at the first unread element. If there are no subsequent READ statements, the extra data is ignored.

READ
Statement

To reread data from any line in the list of DATA statements, use the RESTORE statement (see "RESTORE Statement" in this chapter).

Example:

```
80 FOR I=1 TO 10
90 READ A(I)
100 NEXT I
110 DATA 3.08,5.19,3.12,3.98,4.24
120 DATA 5.08,5.55,4.00,3.16,3.37
```

This program segment reads the values from the DATA statements into the array A. After execution, the value of A(1) is 3.08, and so on.

```
Ok
10 PRINT "CITY", "STATE", " ZIP"
20 READ C$,S$,Z
30 DATA "DENVER,", COLORADO, 80211
40 PRINT C$,S$,Z
RUN
CITY          STATE         ZIP
DENVER,       COLORADO      80211
Ok
```

This program reads string and numeric data from the DATA statement in line 30. Note that you don't need quotation marks around COLORADO, because it doesn't have commas, semicolons, or significant leading or trailing blanks. However, you do need the quotation marks around "DENVER" because of the comma.

Purpose: Inserts explanatory remarks in a program.

Versions:

Cassette	Cartridge	Compiler
***	***	(**)

Format: REM *remark*

Remarks: *remark* may be any sequence of characters.

REM statements are not executed but are output exactly as entered when the program is listed. However, they do slow up execution time somewhat, and take up space in memory.

REM statements may be branched into (from a GOTO or GOSUB statement), and execution continues with the first executable statement after the REM statement.

Remarks may be added to the end of a line by preceding the remark with a single quotation mark instead of :REM. If you put a remark on a line with other BASIC statements, the remark must be the *last* statement on the line.

Example:

```
100 REM calculate average velocity
110 SUM=0: REM initialize SUM
```

Line 110 might also be written:

```
110 SUM=0 ' initialize SUM
```

RENUM
Command

Purpose: Renumbers program lines.

Versions: Cassette *** Cartridge *** Compiler

Cassette	Cartridge	Compiler
***	***	

Format: RENUM [*newnum*] [,[*oldnum*] [,*increment*]]

Remarks:

newnum is the first line number to be used in the new sequence. The default is 10.

oldnum is the line in the current program where renumbering is to begin. The default is the first line of the program.

increment is the increment to be used in the new sequence. The default is 10.

RENUM also changes all line number references following GOTO, GOSUB, THEN, ELSE, ON...GOTO, ON...GOSUB, RESTORE, RESUME, and ERL test statements to reflect the new line numbers. If a nonexistent line number appears after one of these statements, the error message **Undefined line number xxxxx in yyyyy** is printed. The incorrect line number reference (xxxxx) is not changed by RENUM, but line number yyyyy may be changed.

> **Note:** RENUM cannot be used to change the order of program lines (for example, RENUM 15,30 when the program has three lines numbered 10, 20 and 30) or to create line numbers greater than 65529. An **Illegal function call** error results.

Example:

```
RENUM
```

Renumbers the entire program. The first new line number is 10. Lines increment by 10.

```
RENUM 300,,50
```

Renumbers the entire program. The first new line number is 300. Lines increment by 50.

```
RENUM 1000,900,20
```

Renumbers the lines from 900 up so they start with line number 1000 and increment by 20.

RESET
Command

Purpose: Closes all diskette files and clears the system buffer.

> **Note:** This command requires the use of DOS 2.10. If DOS 2.10 is not present, an **Illegal function call** error will occur.

Versions:

Cassette	Cartridge	Compiler
	***	***

Format: RESET

Remarks: If all open files are on diskette, then RESET is the same as CLOSE with no file numbers after it.

Purpose: Allows DATA statements to be reread from a specified line.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: RESTORE [*line*]

Remarks:

line is the line number of a DATA statement in the program.

After a RESTORE statement is executed, the next READ statement accesses the first item in the first DATA statement in the program. If *line* is specified, the next READ statement accesses the first item in the specified DATA statement.

Example:

```
Ok
10 READ A,B,C
20 RESTORE
30 READ D,E,F
40 DATA 57, 68, 79
50 PRINT A;B;C;D;E;F
RUN
 57  68  79  57  68  79
Ok
```

The RESTORE statement in line 20 resets the DATA pointer to the beginning, so that the values that are read in line 30 are 57, 68, and 79.

RESUME
Statement

Purpose: Continues program execution after an error recovery procedure is performed.

Versions:

Cassette	Cartridge	Compiler
***	***	(**)

Format: RESUME [0]

RESUME NEXT

RESUME *line*

Remarks: Any of the formats shown above may be used, depending on where execution is to resume:

RESUME or RESUME 0
Execution resumes at the statement which caused the error.

> **Note:** If you try to renumber a program containing a RESUME 0 statement, you will get an **Undefined line number** error. The statement will still say RESUME 0, which is okay.

RESUME NEXT Execution resumes at the statement immediately following the one which caused the error.

RESUME *line* Execution resumes at the specified line number.

A RESUME statement that is not in an error trap routine causes a **RESUME without error** message to occur.

Example:

```
10 ON ERROR GOTO 900
.
.
.
900 IF (ERR=230)AND(ERL=90) THEN PRINT
          "TRY AGAIN": RESUME 80
```

Line 900 is the beginning of the error trapping routine. The RESUME statement causes the program to return to line 80 when error 230 occurs in line 90.

RETURN Statement

Purpose: To bring you back from a subroutine. See "GOSUB and RETURN Statements" in this chapter.

Versions:

Cassette	Cartridge	Compiler
***	***	***

line valid only in Cartridge BASIC and the BASIC Compiler.

Format: RETURN [*line*]

Remarks:

line is the line number of the program line you wish to return to. You may use it only in Cartridge BASIC.

Although you can use RETURN *line* to return from any subroutine, this enhancement was added to allow non-local returns from the event trapping routines. From one of these routines you will often want to go back to the BASIC program at a fixed line number while still eliminating the GOSUB entry the trap created. Use of the non-local RETURN must be done with care, however, since any other GOSUB, WHILE, or FOR statements that were active at the time of the trap will remain active.

Purpose: Returns the rightmost *n* characters of string *x$*.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: *v$* = RIGHT$(*x$*,*n*)

Remarks:

x$ is any string expression.

n is an integer expression specifying the number of characters to be in the result.

If *n* is greater than or equal to LEN(*x$*), then *x$* is returned. If *n* is zero, the null string (length zero) is returned.

Also see the MID$ and LEFT$ functions.

Example:

```
Ok
10 A$="BOCA RATON, FLORIDA"
20 PRINT RIGHT$(A$,7)
RUN
FLORIDA
Ok
```

The rightmost seven characters of the string A$ are returned.

RMDIR Command

Purpose: Removes a directory from the specified diskette.

Note: This command requires the use of DOS 2.10.

Versions: Cassette Cartridge Compiler

Cassette	Cartridge	Compiler
	***	

Format: RMDIR *path*

Remarks:

path is a string expression, not exceeding 63 characters, that identifies the sub-directory to be removed from the existing directory. Refer to "Naming Files" and "Tree-Structured Directories" in Chapter 3 for more information.

The directory must be empty of all files and sub-directories before it can be removed, with the exception of the "." and ".." entries, or a **Path/file access error** occurs.

Examples:

```
              ROOT
              /  \
             /    \
        SALES    ACCOUNTING
         /        /      \
        /        /        \
     MIKE     SHANNON    CHELLE
      /
     /
   PAM
```

The following examples refer to the tree-structure above.

If you are in the root directory and you want to remove the directory called PAM, use:

```
RMDIR  "SALES\MIKE\PAM"
```

If you want to make ACCOUNTING the current directory and remove the directory called CHELLE, use:

```
CHDIR "ACCOUNTING"
RMDIR "CHELLE"
```

Another way to remove the directory CHELLE is to make the root the current directory and then remove CHELLE.

```
CHDIR "\"
RMDIR "ACCOUNTING\CHELLE"
```

The directory preceding the current directory cannot be removed. Using the tree-structure above, suppose that MIKE is the current directory. If you try to remove the SALES directory you will get a **Path/file access error**.

If you try to use the KILL command to remove a directory, you will get a **Path/file access error**.

Purpose: Returns a random number between 0 and 1.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: v = RND[(x)]

Remarks:

x is a numeric expression which affects the returned value as described below.

The same sequence of random numbers is generated each time the program is run unless the random number generator is reseeded. This is most easily done using the RANDOMIZE statement (see "RANDOMIZE Statement" in this chapter). You may also reseed the generator when you call the RND function by using *x* where *x* is negative. This always generates the particular sequence for the given *x*. This sequence is not affected by RANDOMIZE, so if you want it to generate a different sequence each time the program is run, you must use a different value for *x* each time.

If *x* is positive or not included, RND(*x*) generates the next random number in the sequence.

RND(0) repeats the last number generated.

To get random numbers in the range 0 (zero) through *n*, use the formula:

INT(RND * (*n*+1))

RND
Function

Example: The first horizontal line of results shows three random numbers, generated using a positive *x*.

In line 40, a negative number is used to reseed the random number generator. The random numbers produced after this reseeding are in the second row of results.

Line 80 uses RANDOMIZE to reseed the random number generator; in line 90 it is reseeded again by calling RND with the same negative value we used in line 40. This cancels the effect of the RANDOMIZE statement, as you can see; the third line of results is identical to the second line.

In line 130, RND is called with an argument of zero, so the last number printed is the same as the preceding number.

```
Ok
10 FOR I=1 TO 3
20 PRINT RND(I);      ' x>0
30 NEXT I
40 PRINT: X=RND(-6) ' x<0
50 FOR I=1 TO 3
60 PRINT RND(I);      ' x>0
70 NEXT I
80 RANDOMIZE 853 'randomize
90 PRINT: X=RND(-6) ' x<0
100 FOR I=1 TO 3
110 PRINT RND;     ' same as x>0
120 NEXT I
130 PRINT: PRINT RND(0)
RUN
 .6291626  .1948297  .6305799
 .6818615  .4193624  .6215937
 .6818615  .4193624  .6215937
 .6215937
Ok
```

Purpose: Begins execution of a program.

Versions:

Cassette	Cartridge	Compiler
***	***	(**)

Format: RUN [*line*]

RUN *filespec*[,R]

Remarks:

line is the line number of the program in memory where you wish execution to begin.

filespec is a string expression for the file specification, as explained under "Naming Files" in Chapter 3. The default extension .BAS is supplied for diskette files.

RUN or RUN *line* begins execution of the program currently in memory. If *line* is specified, execution begins with the specified line number. Otherwise, execution begins at the lowest line number.

RUN *filespec* loads a file from diskette or cassette into memory and runs it. It closes all open files and deletes the current contents of memory before loading the designated program. However, with the **R** option, all data files remain open. Refer also to Appendix B, "BASIC Diskette Input and Output."

Executing a RUN command will turn off any sound that is running and reset to music background. All

sound output will be reset to the default SOUND OFF/BEEP ON state. Also, PEN and STRIG will be reset to OFF.

Example:

```
Ok
10 PRINT 1/7
RUN
 .1428571
Ok
10 PI=3.141593
20 PRINT PI
RUN 20
 0
Ok
```

In this first example, we use the first form of RUN on two very small programs. The first program is run from the beginning. We used the RUN *line* option for the second example to run the program from line 20. In this case, line 10 does not get executed, so PI does not receive its proper value. A 0 is printed because all numeric variables have an initial value of zero.

```
RUN "CAS1:NEWFIL",R
```

The preceding example loads the program "NEWFIL" from the tape and runs it, keeping files open.

Purpose: Saves a BASIC program file on diskette or cassette.

Versions:

Cassette	Cartridge	Compiler
***	***	

Format: SAVE *filespec* [,A]

SAVE *filespec* [,P]

Remarks:

filespec is a string expression for the file specification. If *filespec* does not conform to the rules outlined under "Naming Files" in Chapter 3, an error is issued and the save is canceled.

The BASIC program is written to the specified device. When saving to CAS1:, the cassette motor is turned on and the file is immediately written to the tape.

For diskette files, if the filename is eight characters or less and no extension is supplied, the extension **.BAS** is added to the name. If a file with the same filename already exists on the diskette, it will be written over.

When using Cassette BASIC and Cartridge BASIC without DOS, if the device name is omitted, CAS1: is assumed. CAS1: is the only allowable device for SAVE in Cassette BASIC.

For Cartridge BASIC, the device defaults to the DOS default drive when DOS is present or else the device defaults to cassette.

The **A** option saves the program in ASCII format. Otherwise, BASIC saves the file in a compressed binary (tokenized) format. ASCII files take up more space, but some types of access require that files be in ASCII format. For example, a file intended to be merged must be saved in ASCII format. Programs saved in ASCII may be read as data files.

The **P** option saves the program in an encoded binary format. This is the protection option. When a protected program is later run (or loaded), any attempt to LIST or EDIT it fails with an **Illegal function call** error. No way is provided to "unprotect" such a program.

> **Note:** The diskette directory entry for a BASIC program file gives no indication that the file is either protected or stored in ASCII format. The **.BAS** extension is used in any case.

See also Appendix B, "BASIC Diskette Input and Output."

Example:

```
SAVE "INVENT"
```

Saves the program in memory as INVENT. The program is saved on cassette if you are using Cassette BASIC. If you are using Cartridge BASIC, and you are using DOS, the program is saved on the diskette in the DOS default drive and given an extension of **.BAS**.

```
SAVE "A:PROG",A
```

Saves PROG.BAS on drive A in ASCII, so it can be merged later.

```
SAVE "A:SECRET.BOZ",P
```

Saves SECRET.BOZ on drive A, protected so it may not be altered.

SCREEN Function

Purpose: Returns the ASCII code (0-255) for the character on the active screen at the specified row (line) and column.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: *v* = SCREEN(*row,col*[,*z*])

Remarks:

row is a numeric expression in the range 1 to 25.

col is a numeric expression in the range 1 to 20, 1 to 40 or 1 to 80 depending on the WIDTH setting.

z is a numeric expression which evaluates to a true or false value. *z* is only valid in text mode.

Refer to Appendix G, "ASCII Character Codes" for a list of ASCII codes.

In text mode, if *z* is included and is true (non-zero), the color attribute for the character is returned instead of the code for the character. The color attribute is a number in the range 0 to 255. This number, *v*, may be explained as follows:

(*v* MOD 16) is the foreground color.

(((*v* - *foreground*)/16) MOD 128) is the background color, where *foreground* is calculated as above.

(v>127) is true (-1) if the character is blinking, false (0) if not.

Refer to "COLOR Statement" for a list of colors and their associated numbers.

In graphics mode, if the specified location contains graphic information (points or lines, as opposed to just a character), then the SCREEN function returns zero.

Any values entered outside the ranges indicated result in an **Illegal function call** error.

The SCREEN *statement* is explained in the next section.

Example:

```
100 X = SCREEN (10,10)
```

If the character at 10,10 is A, then X is 65.

```
110 X = SCREEN (1,1,1)
```

Returns the color attribute of the character in the upper left hand corner of the screen.

SCREEN Statement

Purpose: Sets the screen attributes to be used by subsequent statements.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: SCREEN [*mode*] [,[*burst*] [,[*apage*] [,*vpage*]] [,*erase*]]

Remarks:

mode is a numeric expression resulting in an integer value in the range of 0 through 6. Valid modes are:

0 Text mode at current WIDTH (40 or 80). 2K page size when using WIDTH 40, and 4K page size when using WIDTH 80. WIDTH 80 is available only if you have 128K of memory.

1 Medium resolution graphics mode (320x200). 4 colors. 16K page size.

2 High resolution graphics mode (640x200). 2 colors. 16K page size.

3 Low resolution graphics mode (160x200). 16 colors. 16K page size. Supported in Cartridge BASIC only.

4 Medium resolution graphics mode (320x200). 4 colors. 16K page size. Supported in Cartridge BASIC only.

5 Medium resolution graphics mode (320x 200). 16 colors. 32K page size. Available only if you have 128K of memory. Supported in Cartridge BASIC only.

6 High resolution graphics mode (640x200). 4 colors. 32K page size. Available only if you have 128K of memory. Supported in Cartridge BASIC only.

burst is a numeric expression resulting in either a 0 or a 1. It enables or disables color. On an RGB monitor, color burst is always on. On a composite monitor, color burst may be on or off. In text mode (mode=0), a zero value disables color (grey scale images only) and a 1 value enables color (allows color images). In medium resolution graphics (mode=1 or mode=4), a 0 value enables color and a 1 value disables color. Burst off for mode=1 and mode=4 also produces a grey scale. Color burst has no effect for mode=2, mode=3, mode=5, or mode=6. Color burst is always on for these modes.

apage (active page) is an integer expression in the range of 0 to n determined by current video memory size and page size for current screen mode. It selects the page to be written to by output statements to the screen. *apage* is valid in all screen modes. If omitted, the default is the current active page.

vpage (visual page) selects which page is to be displayed on the screen, in the same way

as *apage* above. The visual page may be different from the active page. *vpage* is valid in all screen modes. If omitted, *vpage* defaults to *apage*.

erase is an integer expression in the range of 0-2 indicating how much or how little of video memory should be erased.

0 Do not erase video memory even if the mode changes.

1 Erase the union of the new page and the old page if the mode or burst changes.

2 Erase all of video memory if mode or burst changes. The default is 1.

erase is only supported in Cartridge BASIC.

Mode	Desc	Width	Psize	Colors
0	Alpha	40,80	2k 4k	16
1	320x200	40	16k	4
2	640x200	80	16k	2
3	160x200	20	16k	16

Mode	Desc	Width	Psize	Colors
4	320x200	40	16k	4
5	320x200	40	32k	16
6	640x200	80	32k	4

If all parameters are valid, the new screen mode is stored, the screen is erased, (video memory may or may not be erased) the foreground color is set to white, and the background and border colors are set to black.

If the new screen mode is the same as the previous mode, and the color burst does not change, nothing is changed.

In all modes if *apage* and *vpage* are specified, it is possible to change display pages for viewing. Initially, both active and visual pages default to page 0 (zero). By switching around active and visual pages, you can display one page while building another. Then you can switch visual pages instantaneously.

BASIC forces the visual page to the active page under the following conditions:

1. When BASIC returns to direct mode as a result of an error.

2. When BASIC exits the program via the END statement or by running off the end of the program.

3. When a STOP statement is encountered.

   Note: There is only one cursor shared among all the pages. If you are going to switch active pages back and forth, you should save the cursor

position on the current active page (using POS(0) and CSRLIN), before changing to another active page. Then when you return to the original page, you can restore the cursor position using the LOCATE statement.

Any parameter may be omitted. Omitted parameters, except *vpage*, assume the old value.

Any values entered outside the ranges indicated will result in an **Illegal function call** error. Previous values are kept.

Example: The following program contains a step-by-step explanation of what each line does.

```
10 SCREEN 0,1,0,0: WIDTH 80
```

Selects text mode with color, and sets active and visual page to 0. The page size is 4K.

```
20 SCREEN ,,1,2
```

Mode and color burst remain unchanged. Active page is set to 1 and display page to 2.

```
30 SCREEN 2,,0,0,1
```

Switches to high resolution graphics mode. The union of the old page and current page is erased.

```
40 SCREEN 3,,,,0
```

Switches to low resolution color graphics. Video memory is not erased.

```
50 SCREEN 4,,,,2
```

Sets medium resolution graphics. All of Video memory is erased.

This is an example of how you can have 3 different kinds of print on the screen at the same time by using the *erase* parameter.

```
20  KEY OFF
30  SCREEN 3,1,0,0,2
50  FOR C=15 TO STEP -1
60  COLOR C
70  PRINT "SCREEN 3,1,0,0,2"
80  NEXT C
150 SCREEN 4,1,0,0,0
160 LOCATE 16,1
170 FOR C=3 TO 1 STEP -1
180 COLOR C
190 PRINT "SCREEN 4,1,0,0,0"
200 NEXT C
300 SCREEN 2,1,0,0,0
310 LOCATE 19,1
320 FOR C=1 TO 3
330 PRINT "SCREEN 2,1,0,0,0"
340 NEXT C
350 FOR J=1 TO 2000: NEXT J
360 GOTO 30
```

SGN
Function

Purpose: Returns the sign of *x*.

Versions: Cassette *** Cartridge *** Compiler ***

Cassette	Cartridge	Compiler
***	***	***

Format: *v* = SGN(*x*)

Remarks: *x* is any numeric expression.

SGN(*x*) is the mathematical signum function:

- If *x* is positive, SGN(*x*) returns 1.
- If *x* is zero, SGN(*x*) returns 0.
- If *x* is negative, SGN(*x*) returns -1.

Example:

```
ON SGN(X)+2 GOTO 100,200,300
```

branches to 100 if X is negative, 200 if X is zero, and 300 if X is positive.

Purpose: Calculates the trigonometric sine function.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: *v* = SIN(*x*)

Remarks: *x* is an angle in radians.

If you want to convert degrees to radians, multiply by PI/180, where PI=3.141593.

SIN(*x*) is calculated in single precision in Cassette BASIC and in either single or double precision in Cartridge BASIC.

Example:

```
Ok
10 PI=3.141593
20 DEGREES = 90
30 RADIANS=DEGREES * PI/180 ' PI/2
40 PRINT SIN(RADIANS)
RUN
 1
Ok
```

This example calculates the sine of 90 degrees, after first converting the degrees to radians.

SOUND Statement

Purpose: Produces sound through the speaker.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: SOUND *freq, duration* [,[*volume*][, [*voice*]]]

SOUND ON

SOUND OFF

Remarks:

freq is the desired rate of occurence in Hertz (cycles per second). It must be a numeric expression in the range 37 to 32767. The lowest frequency the multivoice sound chip can produce is 110 Hz. Any values below 110 Hz will sound at 110 Hz. BASIC will not give an error.

duration is the desired length of time in clock ticks. The clock ticks occur 18.2 times per second. The *duration* must be a numeric expression. In Cartridge BASIC, duration is .0015 to 65535.

volume is a numeric expression in the range of 0 to 15. If volume is omitted 15 is assumed. SOUND must be ON or you will get an **Illegal function call** error. Supported in Cartridge BASIC only.

voice is a numeric expression in the range of 0 to 2. If voice is omitted, 0 is assumed.

SOUND must be ON or you get an **Illegal function call** error. Supported in Cartridge BASIC only.

When the SOUND statement produces a sound, the program continues to execute until it reaches another SOUND statement. If *duration* of the new SOUND statement is zero, the current SOUND statement that is running is turned off. Otherwise, the program waits until the first sound completes before it executes the new SOUND statement.

If you are using Cartridge BASIC, you can cause the sounds to be *buffered* so that the program continues to execute even when it comes to a new SOUND statement. See the **MB** command explained under "PLAY Statement" in this chapter for details.

If no SOUND statement is running, SOUND *x*,0 has no effect.

In Cartridge BASIC the SOUND statement can be used with the BEEP statement to send sound to the internal speaker and the external speaker. SOUND ON is used to enable sound to the external speaker which will support multi-voice sound using the PLAY statement or the SOUND statement.

```
SOUND ON
```

Sound will come from the television/external speaker. The internal speaker is disabled. By turning SOUND ON, you can then use multiple voices with the PLAY or SOUND statements and you can control the volume of PLAY or SOUND with the V parameter.

```
SOUND OFF  :  BEEP ON
```

SOUND
Statement

This sends the sound through the television/external speaker and the internal speaker.

```
SOUND OFF  :  BEEP OFF
```

This sends the sound only through the internal speaker.

BASIC will restore the machine to the default BEEP ON/SOUND OFF state when it exits.

Refer to the "PLAY Statement" in this chapter for an explanation of PLAY with multiple voices and volume.

The tuning note, A, has a frequency of 440. The following table correlates notes with their frequencies for two octaves on either side of middle C.

Note	Frequency	Note	Frequency
C	130.810	C*	523.250
D	146.830	D	587.330
E	164.810	E	659.260
F	174.610	F	698.460
G	196.000	G	783.990
A	220.000	A	880.000
B	246.940	B	987.770
C	261.630	C	1046.500
D	293.660	D	1174.700
E	329.630	E	1318.500
F	349.230	F	1396.900
G	392.000	G	1568.000
A	440.000	A	1760.000
B	493.880	B	1975.500

* middle C. Higher (or lower) notes may be approximated by doubling (or halving) the frequency of the corresponding note in the previous (next) octave.

To create periods of silence, use SOUND 32767,*duration*.

The time limit for one beat can be calculated from beats per minute by dividing the beats per minute into 1092 (the number of clock ticks in a minute).

The next table shows typical tempos in terms of clock ticks:

SOUND
Statement

	Tempo	Beats/ Minute	Ticks/ Beat
very slow	Larghissimo		
	Largo	40-60	27.3-18.2
	Larghetto	60-66	18.2-16.55
	Grave		
	Lento		
	Adagio	66-76	16.55-14.37
slow	Adagietto		
	Andante	76-108	14.37-10.11
medium	Andantino		
	Moderato	108-120	10.11-9.1
fast	Allegretto		
	Allegro	120-168	9.1-6.5
	Vivace		
	Veloce		
	Presto	168-208	6.5-5.25
very fast	Prestissimo		

Example: The following program creates a glissando up and down in three voices.

```
5 SOUND ON: BEEP OFF
10 FOR I=440 TO 1000 STEP 5
20 SOUND I,0.5,,0
25 SOUND I,0.5,,1
27 SOUND I,0.5,,2
30 NEXT
40 FOR I=1000 TO 440 STEP -5
50 SOUND I,0.5,3,0
60 SOUND I,0.5,6,1
70 SOUND I,0.5,9,2
80 NEXT
```

Purpose: Returns a string consisting of *n* spaces.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: *v$* = SPACE$(*n*)

Remarks: *n* must be in the range 0 to 255.

Refer also to the SPC function.

Example:

```
Ok
10 FOR I = 1 TO 5
20 X$ = SPACE$(I)
30 PRINT X$;I
40 NEXT I
RUN
 1
   2
    3
     4
      5
Ok
```

This example uses the SPACE$ function to print each number I on a line preceded by I spaces. An additional space is inserted because BASIC puts a space in front of positive numbers.

STATEMENTS

SPC Function

Purpose: Skips *n* spaces in a PRINT statement.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: PRINT SPC(*n*)

Remarks: *n* must be in the range 0 to 255.

If *n* is greater than the defined width of the device, then the value used is *n* MOD *width*. SPC may only be used with PRINT, LPRINT and PRINT # statements.

If the SPC function is at the end of the list of data items, then BASIC does not add a carriage return, as though the SPC function had an implied semicolon after it.

Also see the SPACE$ function.

Example:

```
Ok
PRINT "OVER" SPC(15) "THERE"
OVER               THERE
Ok
```

This example prints OVER and THERE separated by 15 spaces.

Purpose: Returns the square root of *x*.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: *v* = SQR(*x*)

Remarks: *x* must be greater than or equal to zero.

SQR is calculated in single precision in Cassette BASIC and in either single or double precision in Cartridge BASIC.

Example:

```
Ok
10 FOR X = 10 TO 25 STEP 5
20 PRINT X, SQR(X)
30 NEXT
RUN
 10            3.162278
 15            3.872984
 20            4.472136
 25            5
Ok
```

This example calculates the square roots of the numbers 10, 15, 20 and 25.

STICK
Function

Purpose: Returns the x and y coordinates of two joysticks.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: *v* = STICK(*n*)

Remarks:

n is a numeric expression in the range 0 to 3 which affects the result as follows:

0 returns the x coordinate for joystick A.

1 returns the y coordinate of joystick A.

2 returns the x coordinate of joystick B.

3 returns the y coordinate of joystick B.

Note: STICK(0) retrieves all four values for the coordinates, and returns the value for STICK(0). STICK(1), STICK(2), and STICK(3) do not sample the joy stick. They get the values previously retrieved by STICK(0).

Then range of values for x and y depends on your particular joysticks.

Example:

```
10 PRINT "Joystick B"
20 PRINT "x coordinate","y coordinate"
30 FOR J=1 TO 100
40 TEMP=STICK(0)
50 X=STICK(2): Y=STICK(3)
60 PRINT X,Y
70 NEXT
```

This program takes 100 samples of the coordinates of joystick B and prints them.

STOP Statement

Purpose: Terminates program execution and returns to command level.

Versions:

Cassette	Cartridge	Compiler
***	***	(**)

Format: STOP

Remarks: STOP statements may be used anywhere in a program to terminate execution. When BASIC encounters a STOP statement, it displays the following message:

```
Break in nnnnn
```

where nnnnn is the line number where the STOP occurred.

Unlike the END statement, the STOP statement does not close files.

BASIC always returns to command level after it executes a STOP. You can resume execution of the program by issuing a CONT command (see "CONT Command" in this chapter).

Example:

```
10 INPUT A, B
20 TEMP= A*B
30 STOP
40 FINAL = TEMP+200: PRINT FINAL
RUN
? 26, 2.1
Break in 30
Ok
PRINT TEMP
 54.6
Ok
CONT
 254.6
Ok
```

This example calculates the value of TEMP, then stops. While the program is stopped, we can check the value of TEMP. Then we can use CONT to resume program execution at line 40.

STR$
Function

Purpose: Returns a string representation of the value of *x*.

Versions: Cassette *** Cartridge *** Compiler ***

Format: *v$* = STR$(*x*)

Remarks: *x* is any numeric expression.

If *x* is positive, the string returned by STR$ contains a leading blank (the space reserved for the plus sign). For example:

```
Ok
? STR$(321); LEN(STR$(321))
 321 4
Ok
```

The VAL function is complementary to STR$.

Example: This example branches to different sections of the program based on the number of digits in a number that is entered. The digits in the number are counted by using STR$ to convert the number to a string, then branching based on the length of the string.

```
5 REM arithmetic for kids
10 INPUT "TYPE A NUMBER";N
20 ON LEN(STR$(N))-1 GOSUB 30,100,200,300
.
.
.
```

Purpose: Returns the status of the joystick buttons (triggers).

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: As a statement:

STRIG ON

STRIG OFF

As a function:

v = STRIG(*n*)

Remarks:

n is a numeric expression in the range 0 to 3. It affects the value returned by the function as follows:

0 Returns -1 if button A1 was pressed since the last STRIG(0) function call, returns 0 if not.

1 Returns -1 if button A1 is currently pressed, returns 0 if not.

2 Returns -1 if button B1 was pressed since the last STRIG(2) function call, returns 0 if not.

3 Returns -1 if button B1 is currently pressed, returns 0 if not.

STRIG
Statement and Function

In Cartridge BASIC and the BASIC Compiler, you can read four buttons from the joy sticks. The additional values for *n* are:

4 Returns -1 if button A2 was pressed since the last STRIG(4) function call, returns 0 if not.

5 Returns -1 if button A2 is currently pressed, returns 0 if not.

6 Returns -1 if button B2 was pressed since the last STRIG(6) function call, returns 0 if not.

7 Returns -1 if button B2 is currently pressed, returns 0 if not.

STRIG ON must be executed before any STRIG(*n*) function calls may be made. After STRIG ON, every time the program starts a new statement BASIC checks to see if a button has been pressed.

If STRIG is OFF, no testing takes place.

Refer also to the next section, "STRIG(n) Statement" for enhancements to the STRIG function in Cartridge BASIC.

Purpose: Enables and disables trapping of the joystick buttons.

Versions:

Cassette	Cartridge	Compiler
	***	(**)

Format: STRIG(*n*) ON

STRIG(*n*) OFF

STRIG(*n*) STOP

Remarks:

n may be 0, 2, 4, or 6, and indicates the button to be trapped as follows:

0	button A1
2	button B1
4	button A2
6	button B2

STRIG(*n*) ON must be executed to enable trapping by the ON STRIG(*n*) statement (see "ON STRIG(n) Statement" in this chapter). After STRIG(*n*) ON, every time the program starts a new statement, BASIC checks to see if the specified button has been pressed.

If STRIG(*n*) OFF is executed, no testing or trapping takes place. Even if the button is pressed, the event is not remembered.

If a STRIG(*n*) STOP statement is executed, no trapping takes place. However, if the button is pressed it is remembered so that an immediate trap takes place when STRIG(*n*) ON is executed.

STRIG(n) Statement

Refer also to the previous section, "STRIG Statement and Function."

Purpose: Returns a string of length *n* whose characters all have ASCII code *m* or the first character of *x$*.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: *v$* = STRING$(*n*,*m*)

*v$* = STRING$(*n*,*x$*)

Remarks:

n, m are in the range 0 to 255.

x$ is any string expression.

Example: The first example repeats an ASCII value of 45 to print a string of hyphens.

```
Ok
10 X$ = STRING$(10,45)
20 PRINT X$ "MONTHLY REPORT" X$
RUN
----------MONTHLY REPORT----------
Ok
```

The second example repeats the first character of the string “ABCD.”

```
Ok
10 X$="ABCD"
20 Y$=STRING$(10,X$)
30 PRINT Y$
RUN
AAAAAAAAAA
Ok
```

SWAP Statement

Purpose: Exchanges the values of two variables.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: SWAP *variable1, variable2*

Remarks:

variable1, variable2
are the names of two variables or array elements.

Any type variable may be swapped (integer, single-precision, double-precision, string), but the two variables must be of the same type or a **Type mismatch** error results.

Example:

```
Ok
10 A$=" ONE " : B$=" ALL " : C$="FOR"
20 PRINT A$ C$ B$
30 SWAP A$, B$
40 PRINT A$ C$ B$
RUN
 ONE FOR ALL
 ALL FOR ONE
Ok
```

After line 30 is executed, A$ has the value " ALL " and B$ has the value " ONE ."

Purpose: Exits BASIC and returns to DOS.

> **Note:** This command requires the use of DOS 2.10. If DOS 2.10 is not present then an **Illegal function call** error will occur.

Versions:

Cassette	Cartridge	Compiler
	***	***

Format: SYSTEM

Remarks: SYSTEM closes all files before it returns to DOS. Your BASIC program is lost.

If you entered BASIC through a Batch file from DOS, the SYSTEM command returns you to the Batch file, which continues executing at the point it left off.

TAB
Function

Purpose: Tabs to position *n*.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: PRINT TAB(*n*)

Remarks: *n* must be in the range 1 to 255.

If the current print position is already beyond space *n*, TAB goes to position *n* on the next line. Space 1 is the leftmost position, and the rightmost position is the defined WIDTH.

TAB may only be used in PRINT, LPRINT, and PRINT # statements.

If the TAB function is at the end of the list of data items, then BASIC does not add a carriage return, as though the TAB function had an implied semicolon after it. TAB is used in the following example to cause the information on the screen to line up in columns.

Example:

```
Ok
10 PRINT "NAME" TAB(25) "AMOUNT"
20 READ A$,B$
30 PRINT A$ TAB(25) B$
40 DATA "L. M. JACOBS","$25.00"
RUN
NAME                    AMOUNT
L. M. JACOBS            $25.00
Ok
```

Purpose: Returns the trigonometric tangent of *x*.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: *v* = TAN(*x*)

Remarks:

x is the angle in radians. To convert degrees to radians, multiply by PI/180, where PI=3.141593.

TAN(*x*) is calculated in single precision in Cassette BASIC and in either single or double precision in Cartridge BASIC.

Example:

```
Ok
10 PI=3.141593
20 DEGREES=45
30 PRINT TAN(DEGREES*PI/180)
RUN
 1
Ok
```

This example calculates the tangent of 45 degrees.

TERM
Statement

Purpose: Enters Terminal Emulation.

Versions: Cassette Cartridge Compiler

Cassette	Cartridge	Compiler
	***	

Format: TERM

Remarks: The TERM statement is used to load and run a Terminal Emulator program. The Terminal Emulator is a BASIC program that resides in the BASIC Cartridge.

This program supports simple RS232 communication with another computer via the IBM Internal Modem (optional) or through the RS232 Serial Communications port.

When TERM is entered, or executed in a program, the following actions are taken:

1. Any open files are closed.

2. The BASIC workspace is scratched. That is, any BASIC program and data in memory is erased as if a NEW statement were executed.

   If you are in the process of writing a BASIC program, be sure to SAVE it before executing the TERM statement or your program will be lost.

3. The Terminal Emulator BASIC program is loaded into memory and begins execution.

4. The program begins by clearing the screen and displaying the message:

   ```
   (TERM) - Terminal Emulator
   ```

If you have 128k bytes of memory, the screen width is set to 80.

If you have 64k bytes of memory, the screen width is set to 40.

The program then determines if the optional IBM Internal Modem is installed. If so, and if the Line bit rate is set to 300, then it will be used. Otherwise, the RS232 Serial port will be used.

5. The program now continues with the terminal selection menu. The menu lets you select communication parameters.

The terminal selection menu is as follows:

```
(TERM) - Terminal Emulator

1 Line Bit Rate      [300] (300..4800)
2 Data bits          [7] (7 or 8)
3 Parity type        [E] (E,O,N)

4 Host echos         [Y] (Y or N)
5 Screen width       [80] (40 or 80)

6 Modem Command [ ]


Change <line,data>?


f1=Conv f2=Exit f3=Nul f4=Break
```

Setting Terminal parameters

There are 6 parameters which may be altered. To change any of these, enter the line number, followed by a comma (,) followed by the parameter data. The parameter is changed to <data> when the ENTER key is typed.

The current parameter setting is displayed in square brackets ([ ]). A list of valid options for each parameter line is displayed in parentheses following the current value.

If the line number is omitted, or is not in the range 1 to 6, no error is given and the change line you typed is ignored.

If the comma between the line number and data is omitted, no error is given and the change line you typed is ignored.

Typing a line number and comma with no addition data has the effect of erasing the parameter.

If you make a mistake while typing a change, press the Esc key. This erases the line you are typing so you may begin again. No other edit keys are recognized. That is, Ins, Del, Cursor Left, Cursor Right, and Backspace are ignored.

The parameters and their valid options are:

1 Line bit rate	Selects the speed for data transmission. Valid speeds are: 300, 600, 1200, 1800, 2400, and 4800. If the IBM Internal Modem is installed, and the line bit rate is set to 300, then it will be used. For any higher speed, the RS232 Serial Communication Port will be used.

Note: Modem Commands are ignored when the RS232 Serial Communication Port is used.

2 Data bits	Only 7 or 8 data bits may be transmitted and received.
3 Parity type	Selects Even(E), Odd(O), or No(N) parity. Even is most common, some computers may require 8 data bits, no parity.
4 Host echoing	The most common full duplex protocol specifies that characters typed on the keyboard and transmitted will be echoed (sent back when received) by the host computer. The terminal then displays only the characters received from the other computer. This mode of operation is *host echoing of characters*. If characters are not being echoed by the other computer, then they must be displayed on the screen as they are typed. This mode of operation is *local echoing of characters*. The default value for this parameter is Yes (Y), Host echoes characters. If the computer you are connected to does not echo characters (for example, VM/370), then set this parameter to No (N). The

terminal emulator program will then echo all characters as they are typed (local echoing).

5 Screen width	Allows you to set the screen width to 40 or 80 columns. If you have 64k bytes of memory, the width can only be 40. If you have 128k bytes of memory, the width can be 40 or 80.
6 Modem Command	This line is displayed *only* when the IBM Internal Modem is installed in the machine. Any data up to 252 characters may be entered. The data entered for this parameter is sent, as entered, to the Modem when Fn/F1 is pressed (Entering conversational mode). A modem command may be as simple as "DIAL 255-1234" or several commands separated by commas such as "COUNT 3, DIAL 9W555-1234, RETRY." Refer to the Commands section of the IBM Internal Modem in the *Technical Reference* manual for a complete discussion of modem commands and syntax.

Once all of the desired parameters are set, you are ready to communicate with the other computer. You do so by entering "conversational" mode (Fn/F1).

Using the Function keys

Only the first four function keys have meaning in the terminal emulator.

Fn/F1 Conv. Typing Fn/F1 while in the terminal selection menu clears the screen, changes the definition of function key 1 to "Menu," and places the program in conversation mode with the other computer.

If the IBM Internal Modem is installed, the line bit rate is set to 300, and a modem command was specified, then the modem command is sent to the modem. Call progress reporting is now displayed until the other computer answers, or the IBM Internal modem gives up. (Refer to "Call progress reporting" in the IBM Internal modem attachment of the *Technical Reference* manual for a discussion of the messages displayed). You are now communicating with the other computer as a terminal.

If the line bit rate is other than 300, or the IBM Internal Modem is not installed, then the Modem Command is not sent. You are now communicating with the other computer as a terminal.

	Typing Fn/F1 while in conversation mode returns the program to the menu. The communication link is lost. That is, if you are using the IBM Internal modem, it will disconnect the telephone connection (hang up). If you are using the RS232 Serial Communication Port, the line may or may not be disconnected depending on what type of equipment the serial port is connected to (external modem, or other computer).
Fn/F2 Exit	May be pressed at any time. This causes line disconnect (the communication file is closed) and the program returns to BASIC's direct mode. The program remains in memory. You may list it or use it as the basis of a more advanced communication program.
F3 Nul	This allows you to send a null (CHR$(0)) to the other computer. This key is enabled only when you are in conversation mode.
F4 Break	Pressing F4 produces a BREAK signal on the communication line. Some computers may require a break signal in order to stop producing output, or stop a program.

XON/XOFF Protocol

At line bit rates of 1200 or higher, it becomes necessary to suspend character transmission from the other computer long enough to "catch up." This is done by sending the ASCII characters XOFF (Hex 13) and XON (Hex 11) to the other computer. XOFF tells the other computer to suspend transmission, and XON tells it to resume.

The terminal emulator does this for you automatically when the line bit rate is 1200 or above. At speeds below 1200, XON and XOFF are not necessary.

> **Note:** Some hosts (such as VM/370) do not recognize XON and XOFF in this way. If this presents a problem, you may want to redefine these character values in line 100 of the Terminal Emulation program.

TIME$
Variable and Statement

Purpose: Sets or brings back the current time.

> **Note:** This statement requires the use of DOS 2.10. If DOS 2.10 is not present then an Illegal function call error will occur.

Versions:

Cassette	Cartridge	Compiler
	***	***

Format: As a variable:

*v$* = TIME$

As a statement:

TIME$ = *x$*

Remarks: **For the variable (v$ = TIME$):**

The current time is returned as an 8 character string. The string is of the form *hh:mm:ss*, where *hh* is the hour (00 to 23), *mm* is the minutes (00 to 59), and *ss* is the seconds (00 to 59). The time may have been set by DOS before entering BASIC.

For the statement (TIME$ = x$):

The current time is set. *x$* is a string expression indicating the time to be set. *x$* may be given in one of the following forms:

hh Set the hour in the range 0 to 23. Minutes and seconds default to 00.

hh:mm Set the hour and minutes. Minutes must be in the range 0 to 59. Seconds default to 00.

hh:mm:ss Set the hour, minutes, and seconds. Seconds must be in the range 0 to 59.

A leading zero may be omitted from any of the above values, but you must include at least one digit. For example, if you wanted to set the time as a half hour after midnight, you could enter TIME$="0:30," but not TIME$=":30." If any of the values are out of range, an **Illegal function call** error is issued. The previous time is retained. If *x$* is not a valid string, a **Type mismatch** error results.

Example: The following program displays the time continuously in the middle of the screen.

```
10 CLS
20 LOCATE 10,15
30 PRINT TIME$
40 GOTO 30
```

TIMER
Variable

Purpose: Returns a single-precision number representing the number of seconds that have elapsed since midnight or System Reset.

> **Note:** This function requires the use of DOS 2.10. If DOS 2.10 is not present then an **Illegal function call** error will occur.

Versions:

Cassette	Cartridge	Compiler
	***	

Format: *v*=TIMER

Remarks: Fractional seconds are calculated to the nearest degree possible. TIMER is a read-only variable.

Example:

```
Ok
20 TIME$="23:59:59"
30 FOR I=1 TO 20
40 PRINT "TIME$= ";TIME$,"TIMER=";TIMER
50 NEXT
RUN

TIME$= 23:59:59        TIMER= 86399.06
TIME$= 23:59:59        TIMER= 86399.11
TIME$= 23:59:59        TIMER= 86399.18
             .
             .
             .
TIME$= 24:00:00        TIMER= 0
TIME$= 00:00:00        TIMER= .05
TIME$= 00:00:00        TIMER= .16
TIME$= 00:00:00        TIMER= .21
Ok
```

TRON and TROFF Commands

Purpose: Traces the execution of program statements.

Versions:

Cassette	Cartridge	Compiler
***	***	(**)

Format: TRON

TROFF

Remarks: As an aid in debugging, the TRON command (which may be entered in indirect mode) enables a trace flag that prints each line number of the program as it is executed. The numbers appear enclosed in square brackets. The trace is turned off by the TROFF command.

Example:

```
Ok
10 K=10
20 FOR J=1 TO 2
30 L=K + 10
40 PRINT J;K;L
50 K=K+10
60 NEXT
70 END
TRON
Ok
RUN
[10][20][30][40] 1 10 20
[50][60][30][40] 2 20 30
[50][60][70]
Ok
TROFF
Ok
```

This example uses TRON and TROFF to trace execution of a loop. The numbers in brackets are

STATEMENTS

line numbers; the numbers not in brackets at the end of each line are the values of J, K, and L which are printed by the program.

Purpose: Calls the indicated machine language subroutine with the argument *arg*.

Versions:

Cassette	Cartridge	Compiler
***	***	(**)

Format: *v* = USR[*n*](*arg*)

Remarks:

n is in the range 0 to 9 and corresponds to the digit supplied with the DEF USR statement for the desired routine (see "DEF USR Statement" in this chapter). If *n* is omitted, USR0 is assumed.

arg is any numeric expression or string variable, which will be the argument to the machine language subroutine.

The CALL statement is another way to call a machine language subroutine. See Appendix C, "Machine Language Subroutines" for complete information on using machine language subroutines.

Example:

```
10 DEF USR0 = &HF000
50 C = USR0(B/2)
60 D = USR0(B/3)
```

The function USR0 is defined in line 10. Line 50 calls the function USR0 with the argument B/2. Line 60 calls USR0 again, with the argument B/3.

VAL
Function

Purpose: Returns the numerical value of string *x$*.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: *v* = VAL(*x$*)

Remarks: *x$* is a string expression.

The VAL function strips blanks, tabs, and line feeds from the argument string to determine the result. For example,

```
VAL("   -3")
```

returns -3.

If the first characters of *x$* are not numeric, then VAL(*x$*) returns 0 (zero).

See the STR$ function for numeric to string conversion.

Example:

```
Ok
PRINT VAL("3408 SHERWOOD BLVD.")
 3408
Ok
```

In this example, VAL is used to extract the house number from an address.

VARPTR Function

Purpose: Returns the address in memory of the variable or file control block.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: v = VARPTR(*variable*)

v = VARPTR(#*filenum*)

Remarks:

variable is the name of a numeric or string variable or array element in your program. A value must be assigned to *variable* before the call to VARPTR, or an **Illegal function call** error results.

filenum is the number under which the file was opened.

For both formats, the address returned is an integer in the range 0 to 65535. This number is the offset into BASIC's Data Segment. The address is not affected by the DEF SEG statement.

The first format returns the address of the first byte of data identified with *variable*. The format of this data is described in Appendix I under "How Variables Are Stored."

> **Note:** All simple variables should be assigned before calling VARPTR for an array, because addresses of arrays change whenever a new simple variable is assigned.

VARPTR
Function

VARPTR is usually used to get the address of a variable or array so it may be passed to a USR machine language subroutine. A function call of the form VARPTR(A(0)) is usually specified when passing an array, so that the lowest-addressed element of the array is returned.

The second format returns the starting address of the file control block for the specified file. This is not the same as the DOS file control block. Refer to "BASIC File Control Block" in Appendix I for detailed information about the format of the file control block.

VARPTR is meaningless for cassette files.

Example: This example reads the first byte in the buffer of a random file:

```
10 OPEN "DATA.FIL" AS #1
20 GET #1
30 'get address of control block
40 FCBADR = VARPTR(#1)
50 'figure address of data buffer
60 DATADR = FCBADR+188
70 'get first byte in data buffer
80 A% = PEEK(DATADR)
```

The next example use VARPTR to get the data from a variable. In line 30, P gets the address of the data. Integer data is stored in two bytes, with the less significant byte first. The actual value stored at location P is calculated in line 40. The bytes are read with the PEEK function, and the second byte is multiplied by 256 because it contains the high-order bits.

```
10 DEFINT A-Z
20 DATA1=500
30 P=VARPTR(DATA1)
40 V=PEEK(P) + 256*PEEK(P+1)
50 PRINT V
```

VARPTR$ Function

Purpose: Returns a character form of the address of a variable in memory. It is primarily for use with PLAY and DRAW in programs that will later be compiled.

Versions:

Cassette	Cartridge	Compiler
	***	***

Format: *v$* = VARPTR$(*variable*)

Remarks:

variable is the name of a variable existing in the program.

Note: All simple variables should be assigned before calling VARPTR$ for an array element, because addresses of arrays change whenever a new simple variable is assigned.

VARPTR$ returns a three-byte string in the form:

Byte 0	Byte 1	Byte 2
type	low byte of variable address	high byte of variable address

type indicates the variable type:

2	integer
3	string
4	single-precision
8	double-precision

The value returned by VARPTR$ is the same as:

```
CHR$(type)+MKI$(VARPTR(variable))
```

You can use VARPTR$ to indicate a variable name in the command string for PLAY or DRAW. For example:

Method One	Alternative Method
PLAY "XA$;"	PLAY "X"+VARPTR$(A$)
PLAY "O=I;"	PLAY "O="+VARPTR$(I)

This technique is mainly for use in programs which will later be compiled.

VIEW Statement

Purpose: Allows you to define subsets of the viewing surface, called viewports, onto which window contents are mapped.

Versions:

Cassette	Cartridge	Compiler
	***	

Graphics mode only.

Format: VIEW [[SCREEN] [(*x1,y1*)- (*x2,y2*) [,[*attribute*] [,[*boundary*]]]]]

Remarks:

(x1,y1)-(x2,y2)
are the upper-left (*x1,y1*) and the lower-right (*x2,y2*) coordinates of the viewport defined. The *x* and *y* coordinates must be within the actual limits of the screen or an **Illegal function call** error occurs. For more information, see "Specifying Coordinates" under "Graphics Modes" in Chapter 3.

attribute lets you fill the defined viewport with color assigned to that attribute. If *attribute* is omitted, the viewport is not filled. *Attribute* is in the range of 0 to 15. In low resolution, there are 16 attributes available (0 to 15). In medium resolution, there are 4 (0 to 3) or 16 (0 to 15) attributes available, depending on the current screen mode. In high resolution, there are 2 (0 to 1) or 4 (0 to 3) attributes available, depending on the current

screen mode. The default attribute is always the maximum attribute for the current screen mode. For more information see "Graphics Modes" in Chapter 3.

boundary lets you draw a boundary line around the viewport (if space is available). If *boundary* is omitted, no boundary is drawn. *boundary* can be an attribute in the range 0 to 15 as described in *attribute*.

It is important to note that VIEW sorts the *x* and *y* argument pairs, placing the smallest values for *x* and *y* first. For example:

```
VIEW (100,100)-(5,5)
```

becomes:

```
VIEW (5,5)-(100,100)
```

Another example:

```
VIEW (-4,4)-(4,-4)
```

becomes:

```
VIEW (-4,-4)-(4,4)
```

All possible pairings of *x* and *y* are valid. The only restriction is that *x1* cannot equal *x2* and *y1* cannot equal *y2*. The viewport cannot be larger than the viewing surface.

If the SCREEN argument is omitted, all points plotted are relative to the viewport. That is, *x1* and *y1* are added to the *x* and *y* coordinate before plotting the point on the screen. For example if:

VIEW
Statement

```
10 VIEW (10,10)-(200,100)
```

is executed, then the point plotted by PSET (0,0),3 is at the actual screen location 10,10.

If the SCREEN argument is included, all points plotted are absolute and may be inside or outside of the screen limits. However, only those points that are within the viewport limits are visible. For example if:

```
10 VIEW SCREEN (10,10)-(200,100)
```

is executed, then the point plotted by PSET (0,0),3 does not appear on the screen because 0,0 is outside of the viewport. PSET (10,10),3 is within the viewport and places the point in the upper-left corner.

VIEW with no arguments defines the entire viewing surface as the viewport. This is equivalent to VIEW (0,0)-(159,199) in low resolution, VIEW (0,0)-(319,199) in medium resolution and VIEW (0,0)-(639,199) in high resolution.

You can define multiple viewports, but only one viewport may be active at a time. RUN and SCREEN will disable the viewports.

VIEW allows you to do scaling by changing the size of your viewport. A large viewport will make your objects large and a small viewport will make your objects small. Scaling with VIEW is similar to zooming with WINDOW. (Refer to "WINDOW Statement" in this chapter.)

> **Note:** When using VIEW, the CLS statement will only clear the active current viewport. To clear the entire screen, you must use VIEW to disable the viewport and then use CLS to clear

the screen. Using CLS with viewports does not home the cursor. Ctrl-Home will home the cursor and clear the entire screen.

Examples: The following example defines four viewports:

```
10 SCREEN 1: VIEW: CLS: KEY OFF
20 VIEW (1,1)-(151,91),,1
30 VIEW (165,1)-(315,91),,2
40 VIEW (1,105)-(151,195),,2
50 VIEW (165,105)-(315,195),,1
60 LOCATE 2,4: PRINT "Viewport 1"
70 LOCATE 2,25: PRINT "Viewport 2"
80 LOCATE 15,4: PRINT "Viewport 3"
90 LOCATE 15,25: PRINT "Viewport 4"
100 VIEW (1,1)-(151,91): GOSUB 1000
200 VIEW (165,1)-(315,91): GOSUB 2000
300 VIEW (1,105)-(151,195): GOSUB 3000
400 VIEW (165,105)-(315,195): GOSUB 4000
900 END
1000 CIRCLE (65,50),30,2
1010 'draw a circle in first viewport
1020 RETURN
2000 LINE (45,50)-(90,75),1,B
2010 'draw a line in second viewport
2020 RETURN
3000 FOR D=0 TO 360: DRAW "ta=d;nu20": NEXT
3010 'draw spokes in third viewport
3020 RETURN
4000 PSET(60,50),2: DRAW "e15;f15;l30"
4010 'draw a triangle in fourth viewport
4020 RETURN
```

VIEW
Statement

This example demonstrates scaling with VIEW.

```
10 KEY OFF: CLS: SCREEN 1,0: COLOR 0,0
20 WINDOW SCREEN(320,0)-(0,200)
30 GOTO 140
40 '
50 '========================
60 'PICTURE
70    C=1
80    CIRCLE (160,100),60,C,,,5/18
90    CIRCLE (160,100),60,C,,,1
100 '
110 RETURN
120 '========================
130 '
140 GOSUB 60: FOR I=1 TO 1000: NEXT I: CLS
150 'Create the picture
160 VIEW (1,1)-(160,90),,2: GOSUB 60
170 'Make it small
180 END
```

Purpose: Suspends program execution while monitoring the status of a machine input port.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: WAIT *port, n*[,*m*]

Remarks:

port is the port number, in the range 0 to 65535.

n, m are integer expressions in the range 0 to 255.

Refer to the PCjr *Technical Reference* manual for a description of valid port numbers (I/O addresses).

The WAIT statement causes execution to be suspended until a specified machine input port develops a specified bit pattern.

The data read at the port is XORed with the integer expression *m* and then ANDed with *n*. If the result is zero, BASIC loops back and reads the data at the port again. If the result is nonzero, execution continues with the next statement. If *m* is omitted, it is assumed to be zero.

The WAIT statement lets you test one or more bit positions on an input port. You can test the bit position for either a 1 or a 0. The bit positions to be tested are specified by setting 1's in those positions in *n*. If you do not specify *m*, the input port bits are tested for 1's. If you do specify *m*, a 1 in any bit position in *m* (for which there is a 1 bit in *n*) causes WAIT to test for a 0 for that input bit.

WAIT Statement

When executed, the WAIT statement loops testing those input bits specified by 1's in *n*. If *any one* of those bits is 1 (or 0 if the corresponding bit in *m* is 1), then the program continues with the next statement. Thus WAIT does not wait for an entire pattern of bits to appear, but only for one of them to occur.

> **Note:** It is possible to enter an infinite loop with the WAIT statement. You can do a Fn and Break or a System Reset to exit the loop.

Example: To suspend program execution until port 32 receives a 1 bit in the second bit position:

```
100 WAIT 32,2
```

WHILE and WEND Statements

Purpose: Executes a series of statements in a loop as long as a given condition is true.

Versions:

Cassette	Cartridge	Compiler
***	***	(**)

Format: WHILE *expression*

.
.
.
(*loop statements*)
.
.
.
WEND

Remarks:

expression is any numeric expression.

If *expression* is true (not zero), *loop statements* are executed until the WEND statement is encountered. BASIC then returns to the WHILE statement and checks *expression*. If it is still true, the process is repeated. If it is not true, execution resumes with the statement following the WEND statement.

WHILE–WEND loops may be nested to any level. Each WEND will match the most recent WHILE. An unmatched WHILE statement causes a **WHILE without WEND** error, and an unmatched WEND statement causes a **WEND without WHILE** error.

Example: This example sorts the elements of the string array A$ into alphabetical order. A$ was defined with J elements.

STATEMENTS

WHILE and WEND Statements

```
90 'bubble sort array A$
100 FLIPS=1 'force one pass thru loop
110 WHILE FLIPS
115  FLIPS=0
120  FOR I=1 TO J-1
130   IF A$(I)>A$(I+1) THEN
         SWAP A$(I),A$(I+1): FLIPS=1
140  NEXT I
150 WEND
```

WIDTH Statement

Purpose: Sets the output line width in number of characters. After outputting the indicated number of characters, BASIC adds a carriage return.

Versions:

Cassette	Cartridge	Compiler
***	***	(**)

Format: WIDTH *size*

WIDTH *device,size*

WIDTH #*filenum,size*

Remarks:

size is a numeric expression in the range 0 to 255. This is the new width. WIDTH 0 is the same thing as WIDTH 1.

device is a string expression for the device identifier. Valid devices are SCRN:, LPT1:, COM1:, or COM2:

filenum is a numeric expression in the range 1 to 15. This is the number of a file opened to one of the devices listed below.

Depending on the device specified, the following actions are possible:

WIDTH size or WIDTH "SCRN:",size
Sets the screen width. 20, 40 or 80 column widths are allowed. You must have 128k of memory to use WIDTH 80 or an **Illegal Function call** error will occur. If the screen is in low resolution (screen

3), WIDTH 40 forces the screen into medium resolution. WIDTH 80 forces the screen into high resolution.

If the screen is in medium resolution graphics mode (as would occur with a SCREEN 1 statement), WIDTH 80 forces the screen into high resolution (like a SCREEN 2 statement). WIDTH 20 forces screen into low resolution. (like a SCREEN 3 statement)

If the screen is in high resolution graphics mode (as would occur with a SCREEN 2 statement), WIDTH 40 forces the screen into medium resolution (like a SCREEN 1 statement). WIDTH 20 forces screen into low resolution. (like a SCREEN 3 statement)

> **Note:** Changing the screen width causes the screen to be cleared, and sets the border screen color to black.

WIDTH device,size

Used as a deferred width assignment for the device. This form of width stores the new width value without really changing the current width setting. A subsequent OPEN to the device will use this value for width while the file is open. The width does not change immediately if the device is already open.

> **Note:** LPRINT, LLIST, and LIST,"LPTn:" do an implicit OPEN and are therefore affected by this statement.

WIDTH #filenum,size

The width of the device associated with *filenum* is immediately changed to the new size specified. This allows the width to be changed at will while the file is open. This form of WIDTH has meaning only for LPT1: in Cassette BASIC. Cartridge BASIC also allows COM1: and COM2:. Note that the number sign (#) is required.

Any value entered outside of the ranges indicated will result in an **Illegal function call** error. The previous value is retained.

WIDTH has no effect for the keyboard (KYBD) or cassette (CAS1:).

The width for each printer defaults to 80 when BASIC is started. The maximum width for the IBM 80 CPS Matrix Printer is 132. However, no error is returned for values between 132 and 255.

It is up to you to set the appropriate physical width on your printer. Some printers are set by sending special codes, some have switches. For the IBM 80 CPS Matrix Printer you should use LPRINT CHR$(15); to change to a condensed typestyle when printing at widths greater than 80. Use LPRINT CHR$(18); to return to normal. The IBM 80 CPS Matrix Printer is set up to automatically add a carriage return if you exceed the maximum line length.

Specifying a width of 255 disables line folding. This has the effect of "infinite" width. WIDTH 255 is the default for communications files.

WIDTH
Statement

Changing the width for a communications file does not change either the receive or the transmit buffer; it just causes BASIC to send a carriage return character after every *size* characters.

Changing screen mode affects screen width only when moving between different screen modes such as high resolution to low resolution, high resolution to medium resolution. "SCREEN Statement" in this chapter.

Example:

```
10 WIDTH "LPT1:",75
20 OPEN "LPT1:" FOR OUTPUT AS #1
.
.
.
6020 WIDTH #1,40
```

In the preceding example, line 10 stores a printer width of 75 characters per line. Line 20 opens file #1 to the printer and sets the width to 75 for subsequent PRINT #1,... statements. Line 6020 changes the current printer width to 40 characters per line.

```
SCREEN 1,0   'Set to med-res color graphics
WIDTH 80     'Go to hi-res graphics
WIDTH 40     'Go back to medium res

SCREEN 0,1   'Go to 40x25 text color mode
WIDTH 80     'Go to 80x25 text color mode
```

Purpose: Allows you to redefine the coordinates of the screen.

Versions:

Cassette	Cartridge	Compiler
	***	

Graphics mode only.

Format: WINDOW [[SCREEN] (*x1,y1*)- (*x2,y2*)]

(x1,y1),(x2,y2)
: are programmer defined coordinates called *world coordinates*. These coordinates are single-precision, floating-point numbers. They define the world coordinate space that will be mapped into the the physical coordinate space, as defined by the VIEW statement. (Refer to "VIEW Statement" in this chapter.)

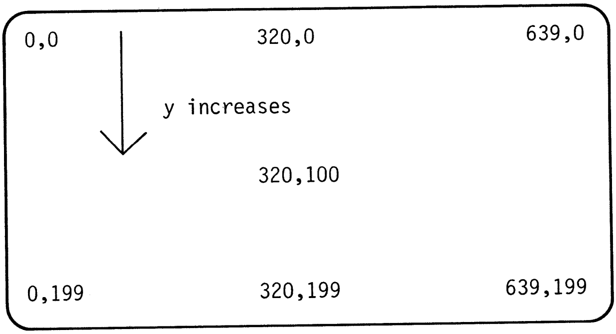
WINDOW allows you to draw objects in space ("world coordinate system") and not be bounded by the logical limits of the screen ("physical coordinate system"). This is done by specifying the world coordinate pairs (*x1,y1*)and (*x2,y2*). This rectangular region in the world coordinate space is called a *window.*

BASIC converts the world coordinate pairs into the appropriate physical coordinate pairs for display within the screen.

In the physical coordinate system, if you enter the following:

```
NEW
SCREEN 2
```

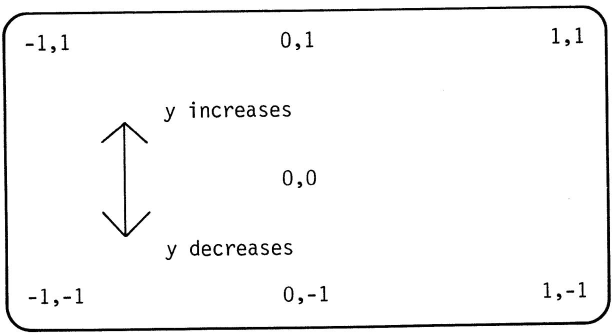
the screen will appear with standard coordinates as:



When the SCREEN attribute is omitted, the screen is viewed in true Cartesian coordinates. For example, given:

```
WINDOW (-1,-1)-(1,1)
```

the screen appears as:

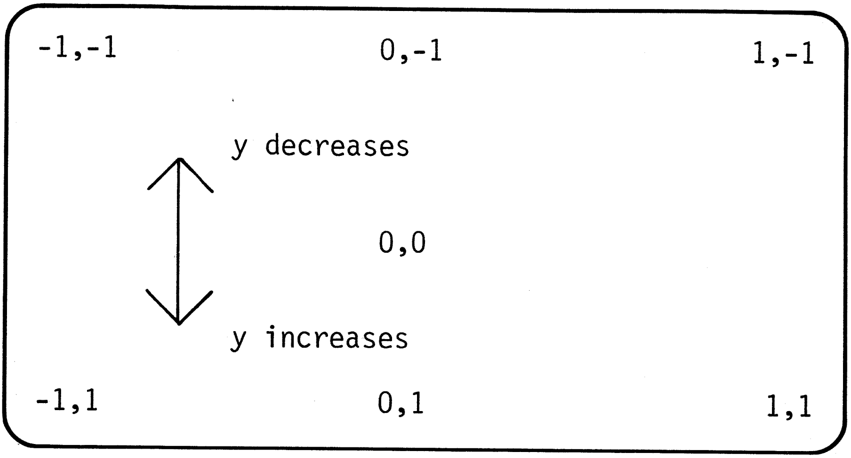


Note that the *y* coordinate is inverted so that *(x1,y1)* is the lower-left coordinate and *(x2,y2)* is the upper-right coordinate.

When the SCREEN attribute is included, the coordinates are not inverted so that *(x1,y1)* is the upper-left coordinate and *(x2,y2)* is the lower-right coordinate. For example:

```
WINDOW SCREEN (-1,-1)-(1,1)
```

defines the screen to look like this:



It is important to note that WINDOW sorts the *x* and *y* argument pairs, placing the smallest values for *x* and *y* first. For example:

```
WINDOW (100,100)-(5,5)
```

becomes:

```
WINDOW (5,5)-(100,100)
```

Another example:

```
WINDOW (-4,4)-(4,-4)
```

becomes:

```
WINDOW (-4,-4)-(4,4)
```

WINDOW
Statement

All possible pairings of *x* and *y* are valid. The only restriction is that *x1* cannot equal *x2* and *y1* cannot equal *y2*.

The WINDOW statement uses *line clipping*, or just "clipping." Clipping is a process in which points referenced outside of a coordinate range are made invisible to the viewing area. Any object lying partially within and partially without a coordinate range is cut off so that only the points referenced in range will appear.

WINDOW also allows you to "zoom" and "pan." Using a window with coordinates larger than an image will display the entire image, but the image will be small and blank spaces will appear on the sides of the screen. Choosing window coordinates smaller than an image forces clipping an allows only a portion of the image to be displayed and magnified. By specifying small and large window sizes, you can zoom in until an object occupies the entire screen, or you can pan out until the image is nothing but a spot on the screen.

RUN, SCREEN, and WINDOW with no attributes will disable any WINDOW coordinates and return the screen to physical coordinates.

Examples: The following example shows clipping using WINDOW.

```
10 SCREEN 2: CLS
20 WINDOW (-6,-6)-(6,6)
30 CIRCLE (4,4),5,1
40 'the circle is large and only part is visible
50 WINDOW (-100,-100)-(100,100)
60 CIRCLE (4,4),5,1 'the circle is very small
70 END
```

The following example shows the effect of zooming using WINDOW.

```
10 KEY OFF: CLS: SCREEN 1,0
20 '
30 GOTO 160
40 '=====================
50 'procedure display
60 '
70 LINE (X,0)-(-X,0),,,&HAA00 'create x axis
80 LINE (0,X)-(0,-X),,,&HAA00 'create y axis
90 '
100 CIRCLE (X/2,X/2),R 'circle has radius r
110 FOR P=1 TO 50:NEXT P 'delay loop
120 '
130 RETURN
140 '====================
150 '
160 X=1000:WINDOW (-X,-X)-(X,X):R=20
170 'create a graph with large coord range
180 GOSUB 50:FOR P=1 TO 1000:NEXT P:CLS
190 '
200 X=60:WINDOW (-X,-X)-(X,X):R=20
210 'smaller coord range increase circle size
220 GOSUB 50:FOR P=1 TO 1000:NEXT P:CLS
230 '
240 X=100:WINDOW (-5,-5)-(X,X):R=20
250 'modify window to show only portion of axes
260 GOSUB 50:FOR P=1 TO 1000:NEXT P:CLS
270 '
280 PRINT "...an illustration of zooming..."
290 FOR P=1 TO 1500:NEXT P
300 CLS:T=-50:U=100:X=U
310 FOR I=1 TO 45
320     T=T + 1:U=U - 1:X=X-1:R=20
330     WINDOW (T,T)-(U,U):CLS:GOSUB 50
340 NEXT I
350 END
```

WRITE Statement

Purpose: Outputs data on the screen.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: WRITE [*list of expressions*]

If the list of expressions is omitted, a blank line is output. If the list of expressions is included, the values of the expressions are output on the screen.

When the values of the expressions are output, each item is separated from the last by a comma. Strings are delimited by quotation marks. After the last item in the list is printed, BASIC adds a carriage return/line feed.

WRITE is similar to PRINT. The difference between WRITE and PRINT is that WRITE inserts commas between the items as they are displayed and delimits strings with quotation marks. Also, positive numbers are not preceded by blanks.

Example: This example shows how WRITE displays numeric and string values.

```
10 A=80: B=90: C$="THAT'S ALL"
20 WRITE A,B,C$
RUN
80,90,"THAT'S ALL"
Ok
```

Purpose: Writes data to a sequential file.

Versions:

Cassette	Cartridge	Compiler
***	***	***

Format: WRITE #*filenum, list of expressions*

Remarks:

filenum is the number under which the file was opened for output.

list of expressions
is a list of string and/or numeric expressions, separated by commas or semicolons.

The difference between WRITE # and PRINT # is that WRITE # inserts commas between the items as they are written and delimits strings with quotation marks. Therefore, it is not necessary for you to put explicit delimiters in the list. Also, WRITE # does not put a blank in front of a positive number. A carriage return/line feed sequence is inserted after the last item in the list is written.

Example: Let A$="CAMERA" and B$="93604-1". The statement:

```
WRITE #1,A$,B$
```

writes the following image to the file.

```
"CAMERA","93604-1"
```

A subsequent INPUT # statement, such as:

```
INPUT #1,A$,B$
```

would input "CAMERA" to A$ and "93604-1" to B$.

Appendixes

Contents

APPENDIXES

APPENDIXES

Notes

Appendix A. Messages

If BASIC detects an error that causes a program to stop running, an error message is displayed. It is possible to trap and test errors in a BASIC program using the ON ERROR statement and the ERR and ERL variables. (For complete explanations of ON ERROR, ERR and ERL, see their explanations in Chapter 4.)

This appendix has two sections. The first section lists alphabetically all of the BASIC error messages with their associated error numbers and includes an explanation of each message. The second section is designed as a quick reference and all message titles are listed in numeric order.

Number	Message
73	**Advanced Feature** Your program used an Advanced BASIC feature while you were using Disk BASIC. Start Advanced BASIC and rerun your program.
54	**Bad file mode** You tried to use PUT or GET with a sequential file or a closed file; or to execute an OPEN with a file mode other than input, output, append, or random. Make sure the OPEN statement was entered and executed properly. GET and PUT require a random file. This error also occurs if you try to merge a file that is not in ASCII format. In this case,

make sure you are merging the right file. If necessary, load the program and save it again using the **A** option.

64 **Bad file name**
An invalid form is used for the filename with KILL, NAME, or FILES.

Check "Naming Files" in Chapter 3 for information on valid filenames, and correct the filename in error.

52 **Bad file number**
A statement uses a file number of a file that is not open, or the file number is out of the range of possible file numbers specified at initialization. Or, the device name in the file specification is too long or invalid, or the filename was too long or invalid.

Make sure the file you wanted was opened and that the file number was entered correctly in the statement. Check that you have a valid file specification (refer to "Naming Files" in Chapter 3 for information on file specifications).

63 **Bad record number**
In a PUT or GET (file) statement, the record number is either greater than the maximum allowed (32767) or equal to zero. In Cartridge BASIC, GET and PUT have been enhanced to allow record numbers in the range 1 to 16,777,215 to accomodate large files with short record numbers.

Correct the PUT or GET statement to use a valid record number.

17 **Can't continue**
You tried to use CONT to continue a program that:

- Halted because of an error,
- Was changed during a break in execution, or
- Does not exist

Make sure the program is loaded, and use RUN to run it.

— **Cartridge Required**
You attempted to run BASIC from a diskette without the BASIC cartridge present under one of the following conditions:

- BASIC 1.x with DOS 1.x
- BASIC 1.x with DOS 2.0
- BASIC 2.0 with DOS 2.0

On a 64K system, attempting to load Advanced BASIC 2.0 with DOS 2.0 gives a PROGRAM TO BIG TO FIT INTO MEMORY error.

With your DOS diskette in the drive, you should first insert the BASIC Cartridge and then load BASIC from diskette.

69 **Communication buffer overflow**
A communication input statement was executed, but the input buffer was already full.

You should use an ON ERROR statement to retry the input when this condition occurs.

Subsequent inputs try to clear this fault unless characters continue to be received faster than the program can process them. If this happens there are several possible solutions:

- Increase the size of the communications buffer using the **/C:** option when you start BASIC.
- Implement a "hand-shaking" protocol with the other computer to tell it to stop sending long enough so you can catch up. (See the example in Appendix F, "Communications.")
- Use a lower baud rate to transmit and receive.

25 **Device Fault**
A hardware error indication was returned by an interface adapter. In Cassette BASIC, this only occurs when a fault status is returned from the printer interface adapter.

This message may also occur when transmitting data to a communications file. In this case, it indicates that one or more of the signals being tested (specified on the OPEN "COM... statement) was not found in the specified period of time.

57 **Device I/O Error**
An error occurred on a device I/O operation. DOS cannot recover from the error.

When receiving communications data, this error can occur from overrun, framing, break, or parity errors. When you are receiving data with 7 or less data bits, the eighth bit is turned on in the byte in error.

24 **Device Timeout**
BASIC did not receive information from an input/output device within a predetermined amount of time. In Cassette BASIC, this only occurs while the program is trying to read from the cassette or write to the printer.

For communications files, this message indicates that one or more of the signals tested with OPEN "COM... was not found in the specified period of time.

Retry the operation.

68 **Device Unavailable**
You tried to open a file to a device which doesn't exist. Either you do not have the hardware to support the device, or you have disabled the device. (For example, you may have used **/C:0** on the BASIC command to start Cartridge BASIC. That would disable communications devices.)

Make sure the device is installed correctly. If necessary, enter the command:

```
SYSTEM
```

This returns you to DOS where you can re-enter the BASIC command.

66 **Direct statement in file**
A direct statement was encountered while loading or chaining to an ASCII format file. The LOAD or CHAIN is terminated.

The ASCII file should consist only of statements preceded by line numbers. This error may occur because of a line feed

character in the input stream. Refer to "Appendix D. Converting Programs to PC jr BASIC."

61 **Disk full**
All diskette storage space is in use. Files are closed when this error occurs.

If there are any files on the diskette that you no longer need, erase them; or, use a new diskette. Then retry the operation or rerun the program.

72 **Disk Media Error**
The controller attachment card detected a hardware or media fault. Usually, this means that the diskette has gone bad.

Copy any existing files to a new diskette and re-format the bad diskette. If formatting fails, the diskette should be discarded.

71 **Disk not Ready**
The diskette drive door is open or a diskette is not in the drive.

Place the correct diskette in the drive and continue the program.

70 **Disk Write Protect**
You tried to write to a diskette that is write-protected.

Make sure you are using the right diskette. If so, remove the write protection, then retry the operation.

This error may also occur because of a hardware failure.

11 **Division by zero**

In an expression, you tried to divide by zero, or you tried to raise zero to a negative power.

It is not necessary to fix this condition, because the program continues running. Machine infinity with the sign of the number being divided is the result of the division; or, positive machine infinity is the result of the exponentiation. This error cannot be trapped.

10 **Duplicate Definition**

You tried to define the size of the same array twice. This may happen in one of several ways:

- The same array is defined in two DIM statements.
- The program encounters a DIM statement for an array after the default dimension of 10 is established for that array.
- The program sees an OPTION BASE statement after an array has been dimensioned, either by a DIM statement or by default.

Move the OPTION BASE statement to make sure it is executed before you use any arrays; or, fix the program so each array is defined only once.

50 **FIELD overflow**

A FIELD statement is attempting to allocate more bytes than were specified for the record length of a random file in the OPEN statement. Or, the end of the FIELD buffer

is encountered while doing sequential I/O (PRINT #, WRITE #, INPUT #) to a random file.

Check the OPEN statement and the FIELD statement to make sure they correspond. If you are doing sequential I/O to a random file, make sure that the length of the data read or written does not exceed the record length of the random file.

58 **File already exists**
The filename specified in a NAME command matches a filename already in use on the diskette.

Retry the NAME command using a different name.

55 **File already open**
You tried to open a file for sequential output or append, and the file is already opened; or, you tried to use KILL on a file that is open.

Make sure you only execute one OPEN to a file if you are writing to it sequentially. Close a file before you use KILL.

53 **File not found**
A LOAD, KILL, NAME, FILES, or OPEN references a file that does not exist on the diskette in the specified drive.

Verify that the correct diskette is in the drive specified, and that the file specification was entered correctly. Then retry the operation.

26 **FOR without NEXT**
A FOR was encountered without a matching NEXT. That is, a FOR loop was active when the physical end of the program was reached.

Correct the program so it includes a NEXT statement.

12 **Illegal direct**
You tried to enter a statement in direct mode which is invalid in direct mode (such as DEF FN).

The statement should be entered as part of a program line.

5 **Illegal function call**
A parameter that is out of range is passed to a system function. The error may also occur as the result of:

- A negative or unreasonably large subscript
- Trying to raise a negative number to a power that is not an integer
- Calling a USR function before defining the starting address with DEF USR
- A negative record number on GET or PUT (file)
- An improper argument to a function or statement
- Trying to list or edit a protected BASIC program
- Trying to delete line numbers which don't exist

Correct the program. Refer to Chapter 4 for information about the particular statement or function.

— **Incorrect DOS version**
The command you just entered requires a different version of DOS from the one you are running.

62 **Input past end**
This is an end of file error. An input statement is executed for a null (empty) file, or after all the data in a sequential file was already input.

To avoid this error, use the EOF function to detect the end of file.

This error also occurs if you try to read from a file that was opened for output or append. If you want to read from a sequential output (or append) file, you must close it and open it again for input.

51 **Internal error**
An internal malfunction occurred in BASIC.

Recopy your diskette. Check the hardware and retry the operation. If the error reoccurs, report to your computer dealer the conditions under which the message appeared.

23 **Line buffer overflow**
You tried to enter a line that has too many characters.

Separate multiple statements on the line so they are on more than one line. You might also use string variables instead of constants where possible.

22 **Missing operand**
An expression contains an operator, such as * or OR, with no operand following it.

Make sure you include all the required operands in the expression.

1 **NEXT without FOR**
The NEXT statement doesn't have a corresponding FOR statement. It may be that a variable in the NEXT statement does not correspond to any previously executed and unmatched FOR statement variable.

Fix the program so the NEXT has a matching FOR.

19 **No RESUME**
The program branched to an active error trapping routine as a result of an error condition or an ERROR statement. The routine does not have a RESUME statement. (The physical end of the program was encountered in the error trapping routine.)

Be sure to include RESUME in your error trapping routine to continue program execution. You may want to add an ON ERROR GOTO 0 statement to your error trapping routine so BASIC displays the message for any untrapped error.

4 **Out of data**
A READ statement is trying to read more data than is in the DATA statements.

Correct the program so that there are enough constants in the DATA statements for all the READ statements in the program.

7 **Out of memory**
A program is too large, has too many FOR loops or GOSUBs, too many variables, expressions that are too complicated, or complex painting.

You may want to use CLEAR at the beginning of your program to set aside more stack space or memory area. See "Tips and Techniques" in Appendix I for further explanation of **Out of memory** errors.

27 **Out of Paper**
The printer is out of paper, or the printer is not switched on.

You should insert paper (if necessary), verify that the printer is properly connected, and make sure that the power is on; then, continue the program.

14 **Out of string space**
BASIC allocates string space dynamically until it runs out of memory. This message means that string variables caused BASIC to exceed the amount of free memory remaining after housecleaning.

6 **Overflow**
The magnitude of a number is too large to be represented in BASIC's number format. Integer overflow will cause execution to stop. Otherwise, machine infinity with the appropriate sign is supplied as the result and execution continues. Integer overflow is the only type of overflow that can be trapped

To correct integer overflow, you need to use smaller numbers, or change to single- or double-precision variables.

Note: If a number is too small to be represented in BASIC's number format, we have an *underflow* condition. If this occurs, the result is zero and execution continues without an error.

75 **Path/file access error**
During an OPEN, RENAME, MKDIR, CHDIR or RMDIR operation, an attempt was made to use a path or filename to an inaccessible file. For example, you tried to open a directory or volume identifier; you tried to open a read only file for writing; or you tried to remove the current directory. The operation is not completed.

76 **Path not found**
During an OPEN, MKDIR, CHDIR or RMDIR operation, DOS is unable to find the path the way it is specified. The operation is not completed.

74 **Rename across disks**
You tried to rename a file, but you specified the incorrect disk. The renaming operation is not performed.

20 **RESUME without error**
The program has encountered a RESUME statement without having trapped an error. The error trapping routine should only be entered when an error occurs or an ERROR statement is executed.

You probably need to include a STOP or END statement before the error trapping routine to prevent the program from "falling into" the error trapping code.

3 **RETURN without GOSUB**
A RETURN statement needs a previous unmatched GOSUB statement.

Correct the program. You probably need to put a STOP or END statement before the subroutine so the program doesn't "fall" into the subroutine code.

16 **String formula too complex**
A string expression is too long or too complex.

The expression should be broken into smaller expressions.

15 **String too long**
You tried to create a string more than 255 characters long.

Try to break the string into smaller strings.

9 **Subscript out of range**
You used an array element either with a subscript that is outside the dimensions of the array, or with the wrong number of subscripts.

Check the usage of the array variable. You may have put a subscript on a variable that is not an array, or you may have coded a built-in function incorrectly.

2 **Syntax error**
A line contains an incorrect sequence of characters, such as an unmatched parenthesis, a misspelled command or statement, or incorrect punctuation. Or, the data in a DATA statement doesn't match the type (numeric or string) of the variable in a READ statement.

When this error occurs, the BASIC program editor automatically displays the line in error. Correct the line or the program.

67 **Too many files**
An attempt is made to create a new file (using SAVE or OPEN) when all directory entries on the diskette are full, or when the file specification is invalid.

If the file specification is okay, use a new formatted diskette and retry the operation.

13 **Type mismatch**
You gave a string value where a numeric value was expected, or you had a numeric value in place of a string value. This error may also be caused by trying to SWAP variables of different types, such as single- and double-precision.

8 **Undefined line number**
A line reference in a statement or command refers to a line which doesn't exist in the program.

Check the line numbers in your program, and use the correct line number.

18 **Undefined user function**
You called a function before defining it with the DEF FN statement.

Make sure the program executes the DEF FN statement before you use the function.

— **Unprintable error**
An error message is not available for the error condition which exists. This is usually caused by an ERROR statement with an undefined error code.

Check your program to make sure you handle all error codes which you create.

30 **WEND without WHILE**

A WEND is encountered before a matching WHILE was executed.

Correct the program so that there is a WHILE for each WEND.

29 **WHILE without WEND**
A WHILE statement does not have a matching WEND. That is, a WHILE was still active when the physical end of the program was reached.

Correct the program so that each WHILE has a corresponding WEND.

Quick Reference

Number	Message
1	NEXT without FOR
2	Syntax error
3	RETURN without GOSUB
4	Out of data
5	Illegal function call
6	Overflow
7	Out of memory
8	Undefined line number
9	Subscript out of range
10	Duplicate Definition
11	Division by zero
12	Illegal direct
13	Type mismatch
14	Out of string space
15	String too long
16	String formula too complex
17	Can't continue
18	Undefined user function
19	No RESUME
20	RESUME without error
22	Missing operand
23	Line buffer overflow
24	Device Timeout
25	Device Fault
26	FOR without NEXT
27	Out of paper
29	WHILE without WEND
30	WEND without WHILE
50	FIELD overflow
51	Internal error

Number	Message
52	Bad file number
53	File not found
54	Bad file mode
55	File already open
57	Device I/O error
58	File already exists
61	Disk full
62	Input past end
63	Bad record number
64	Bad file name
66	Direct statement in file
67	Too many files
68	Device unavailable
69	Communication buffer overflow
70	Disk Write Protect
71	Disk not ready
72	Disk media error
73	Advanced feature
74	Rename across disks
75	Path/file access error
76	Path not found
—	Unprintable error
—	Incorrect DOS Version
—	Cartridge Required

Appendix B. BASIC Diskette Input and Output

This appendix describes procedures and special considerations for using diskette input and output. It contains lists of the commands and statements that are used with diskette files, and explanations of how to use them. Several sample programs are included to help clarify the use of data files on diskette. If you are new to BASIC or if you're getting diskette-related errors, read through these procedures and program examples to make sure you're using all the diskette statements correctly.

You may also want to refer to the IBM Personal Computer *Disk Operating System* manual for other information on handling diskettes and diskette files.

> **Note:** Most of the information in this appendix about program files and sequential files applies to cassette I/O as well. The cassette cannot be opened in random mode, however.

Specifying Filenames

Filenames for diskette files must conform to DOS naming conventions in order for BASIC to be able to read them. Refer to "Naming Files" in Chapter 3 to be sure you are specifying your diskette files correctly.

Commands for Program Files

The commands which you can use with your BASIC program files are listed below, with a quick description. For more detailed information on any of these commands, refer to Chapter 4.

SAVE filespec [,A]

Writes to diskette the program that is currently residing in memory. Optional **A** writes the program as a series of ASCII characters. (Otherwise, BASIC uses a compressed binary format.)

LOAD filespec [,R]

Loads the program from diskette into memory. Optional **R** runs the program immediately. LOAD always deletes the current contents of memory and closes all files before loading. If **R** is included, however, open data files are kept open. Thus, programs can be chained or loaded in sections, and can access the same data files.

RUN filespec [,R]

RUN *filespec* loads the program from diskette into memory and runs it. RUN deletes the current contents of memory

and closes all files before loading the program. If the **R** option is included, however, all open data files are kept open.

MERGE filespec

Loads the program from diskette into memory, but does not delete the current contents of memory. The program line numbers on diskette are merged with the line numbers in memory. If two lines have the same number, only the line from the diskette program is saved. After a MERGE command, the "merged" program resides in memory, and BASIC returns to command level.

KILL filespec

Deletes the file from the diskette.

NAME filespec AS filename

Changes the name of a diskette file.

Protected Files

If you wish to save a program in an encoded binary format, use the **P** (protect) option with the SAVE command. For example:

```
SAVE "MYPROG",P
```

A program saved this way cannot be listed, saved, or edited. Since you cannot "unprotect" such a program, you may also want to save an unprotected copy of the program for listing and editing purposes.

Diskette Data Files - Sequential and Random I/O

Two types of diskette data files may be created and accessed by a BASIC program: sequential files and random access files.

Sequential Files

Sequential files are easier to create than random files but are limited in flexibility and speed when it comes to accessing the data. The data that is written to a sequential file is stored sequentially, one item after another, in the order that each item is sent. Each item is read back in the same way, from the first item in the file, to the last item.

The statements and functions that are used with sequential files are:

CLOSE	WRITE #
INPUT #	EOF
LINE INPUT #	INPUT$
OPEN	LOC
PRINT #	LOF
PRINT # USING	

Creating and Accessing a Sequential File

To create a sequential file and access the data in the file, include the following steps in your program:

1. Open the file for output or append using the OPEN statement.

2. Write data to the file using the PRINT #, WRITE #, or PRINT # USING statements.

3. To access the data in the file, you must close the file (using CLOSE) and reopen it for input (using OPEN).

4. Use the INPUT # or LINE INPUT # statements to read data from the sequential file into the program.

The following are example program lines that demonstrate these steps.

```
100  OPEN "DATA" FOR OUTPUT AS #1 'step 1
200  WRITE #1,A$,B$,C$            'step 2
300  CLOSE #1                     'step 3
400  OPEN "DATA" FOR INPUT AS #1 'also step 3
500  INPUT #1,X$,Y$,Z$            'step 4
```

The above program could also have been written as follows:

```
100  OPEN "O",#1,"DATA"   'step 1
200  WRITE #1,A$,B$,C$    'step 2
300  CLOSE #1             'step 3
400  OPEN "I",#1,"DATA"   'still step 3
500  INPUT #1,X$,Y$,Z$    'step 4
```

Notice that both ways of writing the OPEN statement yield the same results. Look under "OPEN Statement" in Chapter 4 for details of the syntax of each form of OPEN.

The following program, PROGRAM1, is a short program that creates a sequential file, "DATA," from information you enter at the keyboard.

APPENDIXES

Program 1

```
1 REM PROGRAM1 - create a sequential file
10 OPEN "DATA" FOR OUTPUT AS #1
20 INPUT "NAME";N$
25 IF N$="DONE" THEN CLOSE: END
30 INPUT "DEPARTMENT";D$
40 INPUT "DATE HIRED";H$
50 WRITE #1,N$,D$,H$
60 PRINT: GOTO 20
RUN
NAME? MICHELANGELO
DEPARTMENT? AUDIO/VISUAL AIDS
DATE HIRED? 01/12/72

NAME? SHERLOCK HOLMES
DEPARTMENT? RESEARCH
DATE HIRED? 12/03/65

NAME? EBENEEZER SCROOGE
DEPARTMENT? ACCOUNTING
DATE HIRED? 04/27/78

NAME? SUPER MANN
DEPARTMENT? MAINTENANCE
DATE HIRED? 08/16/78

NAME? DONE
Ok
```

Now look at PROGRAM2. It accesses the file "DATA" that was created in PROGRAM1 and displays the name of everyone hired in 1978.

Program 2

```
1 REM PROGRAM2 - accessing a sequential file
10 OPEN "DATA" FOR INPUT AS 1
20 INPUT #1,N$,D$,H$
30 IF RIGHT$(H$,2)="78" THEN PRINT N$
40 GOTO 20
RUN
EBENEEZER SCROOGE
SUPER MANN
Input past end in 20
Ok
```

PROGRAM2 reads, sequentially, every item in the file. When all the data has been read, line 20 causes an **Input past end** error. To avoid getting this error, insert line 15 which uses the EOF function to test for end of file:

```
15 IF EOF(1) THEN CLOSE: END
```

and change line 40 to GOTO 15. The end of file is indicated by a special character in the file. This character has ASCII code 26 (hex 1A). Therefore, you should not put a CHR$(26) in a sequential file.

A program that creates a sequential file can also write formatted data to the diskette with the PRINT # USING statement. For example, the statement:

```
PRINT #1,USING "####.## ";A,B,C,D
```

could be used to write numeric data to diskette without explicit delimiters. The space at the end of the format string serves to separate the items in the diskette file.

The LOC function, when used with a sequential file, returns the number of records that have been written to or read from the file since it was opened. (A record is a 128-byte block of data.) The LOF function returns the number of bytes allocated to the file. This number is always a multiple of 128 (by rounding upward, if necessary) in BASIC 1.10. In Cartridge BASIC, using DOS, records are exact length. They are not rounded to a multiple of 128.

Adding Data to a Sequential File

If you have a sequential file residing on diskette and later want to add more data to the end of it, you cannot simply open the file for output and start writing data. When you open a sequential file for output, you destroy its current contents. Instead, you should open the file for APPEND. Refer to "OPEN Statement" in Chapter 4 for details.

Random Files

Creating and accessing random files requires more program steps than sequential files, but there are advantages to using random files. For instance, numbers in random files are usually stored on diskette in binary formats, while numbers in sequential files are stored as ASCII characters. Therefore, in many cases random files require less space on diskette than sequential files.

The biggest advantage to random files is that data can be accessed randomly; that is, anywhere on the diskette. It is not necessary to read through all the information, as with sequential files. This is possible because the information is stored and accessed in distinct units called records, and each record is numbered.

Records may be any length up to 32767 bytes. The size of a record is not related to the size of a sector on the diskette (512 bytes). BASIC automatically uses all 512 bytes in a sector for information storage. It does this by both blocking records and spanning sector boundaries (that is, part of a record may be at the end of one sector and the other part at the beginning of the next sector).

The statements and functions that are used with random files are:

CLOSE	CVI
FIELD	CVS
GET	LOC
LSET/RSET	LOF
OPEN	MKD$
PUT	MKI$
CVD	MKS$

Creating a Random File

The following program steps are required to create a random file.

1. Open the file for random access. The example which follows to show these steps specifies a record length of 32 bytes. If the record length is omitted, the default is 128 bytes.

2. Use the FIELD statement to allocate space in the random buffer for the variables that will be written to the random file.

3. Use LSET or RSET to move the data into the random buffer. Numeric values must be made into strings when placed in the buffer. To do this, use the "make" functions: MKI$ to make an integer value into a string, MKS$ for a single-precision value, and MKD$ for a double-precision value.

4. Write the data from the buffer to the diskette using the PUT statement.

The following lines show these steps:

```
100  OPEN "FILE" AS #1 LEN=32 'step 1
200  FIELD #1,20 AS N$, 4 AS A$, 8 AS P$
                              'step 2
300  LSET N$=X$               'step 3
400  LSET A$=MKS$(AMT)        'still step 3
500  LSET P$=TEL$             'still step 3
600  PUT #1,CODE%             'step 4
```

Note: Do not use a string variable which has been defined in a FIELD statement in an input statement or on the left side of an assignment (LET) statement. This causes the pointer for that variable to point into string space instead of the random file buffer.

Look at PROGRAM3. It takes information that is entered at the keyboard and writes it to a random file.

Each time the PUT statement is executed, a record is written to the file. The two-digit code that is input in line 30 becomes the record number.

Program 3

```
1 REM PROGRAM3 - create a random file
10 OPEN "FILE" AS #1 LEN=32
20 FIELD #1,20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
35 IF CODE%=99 THEN CLOSE: END
40 INPUT "NAME";X$
50 INPUT "AMOUNT";AMT
60 INPUT "PHONE";TEL$: PRINT
70 LSET N$=X$
80 LSET A$=MKS$(AMT)
90 LSET P$=TEL$
100 PUT #1,CODE%
110 GOTO 30
```

Accessing a Random File

The following program steps are required to access a random file:

1. Open the file for random access.

2. Use the FIELD statement to allocate space in the random buffer for the variables that will be read from the file.

    Note: In a program that performs both input and output on the same random file, you can usually use just one OPEN statement and one FIELD statement.

3. Use the GET statement to move the desired record into the random buffer.

4. The data in the buffer may now be accessed by the program. Numeric values must be converted back

to numbers using the "convert" functions: CVI for integers, CVS for single-precision values, and CVD for double-precision values.

The following program lines show these steps:

```
100 OPEN "FILE" AS 1 LEN=32    'step 1
200 FIELD #1 20 AS N$, 4 AS A$, 8 AS P$
                               'step 2
300 GET #1,CODE%               'step 3
400 PRINT N$                   'step 4
500 PRINT CVS(A$)              'still step 4
```

PROGRAM4 accesses the random file "FILE" that was created in PROGRAM3. By entering the two-digit code at the keyboard, the information associated with that code is read from the file and displayed.

Program 4

```
1 REM PROGRAM4 - access a random file
10 OPEN "FILE" AS 1 LEN=32
20 FIELD #1, 20 AS N$, 4 AS A$, 8 AS P$
30 INPUT "2-DIGIT CODE";CODE%
35 IF CODE%=99 THEN CLOSE: END
40 GET #1, CODE%
50 PRINT N$
60 PRINT USING "$$###.##";CVS(A$)
70 PRINT P$: PRINT
80 GOTO 30
```

The LOC function, with random files, returns the "current record number."The current record number is the last record number that was used in a GET or PUT statement. For example, the statement

```
IF LOC(1)>50 THEN END
```

ends program execution if the current record number in file #1 is higher than 50.

A Sample Program

PROGRAM5 is an inventory program that illustrates random file access. In this program, the record number is used as the part number, and it is assumed the inventory will contain no more than 100 different part numbers. Lines 690-750 initialize the data file by writing CHR$(255) as the first character of each record. This is used later (line 180 and line 320) to determine whether an entry already exists for that part number.

Lines 40-120 display the different inventory functions that the program performs. When you type in the desired function number, line 140 branches to the appropriate subroutine.

Program 5

```
1 REM PROGRAM5 - inventory
120 OPEN "INVEN.DAT" AS #1 LEN=39
125 FIELD #1,1 AS F$,30 AS D$,2 AS Q$,2 AS R$,4 AS P$
130 PRINT: PRINT "OPTIONS:": PRINT
135 PRINT 1,"INITIALIZE FILE"
140 PRINT 2,"CREATE A NEW ENTRY"
150 PRINT 3,"DISPLAY INVENTORY FOR ONE PART"
160 PRINT 4,"ADD TO STOCK"
170 PRINT 5,"SUBTRACT FROM STOCK"
180 PRINT 6,"LIST ITEMS BELOW REORDER LEVEL"
190 PRINT 7,"END APPLICATION"
220 PRINT: PRINT: INPUT "CHOICE";CHOICE
225 IF (CHOICE<1)OR(CHOICE>7) THEN PRINT
      "BAD CHOICE NUMBER": GOTO 130
230 ON CHOICE GOSUB 900,250,390,480,560,680,2000
240 GOTO 220
250 REM build new entry
260 GOSUB 840
270 IF ASC(F$)<>255 THEN INPUT "OVERWRITE";A$:
        IF A$<>"Y" THEN RETURN
280 LSET F$=CHR$(0)
290 INPUT "DESCRIPTION";DESC$
300 LSET D$=DESC$
310 INPUT "QUANTITY IN STOCK";Q%
320 LSET Q$=MKI$(Q%)
330 INPUT "REORDER LEVEL";R%
```

```
340 LSET R$=MKI$(R%)
350 INPUT "UNIT PRICE";P
360 LSET P$=MKS$(P)
370 PUT #1,PART%
380 RETURN
390 REM display entry
400 GOSUB 840
410 IF ASC(F$)=255 THEN PRINT "NULL ENTRY": RETURN
420 PRINT USING "PART NUMBER ###";PART%
430 PRINT D$
440 PRINT USING "QUANTITY ON HAND #####";CVI(Q$)
450 PRINT USING "REORDER LEVEL #####";CVI(R$)
460 PRINT USING "UNIT PRICE $$##.##";CVS(P$)
470 RETURN
480 REM add to stock
490 GOSUB 840
500 IF ASC(F$)=255 THEN PRINT "NULL ENTRY": RETURN
510 PRINT D$: INPUT "QUANTITY TO ADD ";A%
520 Q%=CVI(Q$)+A%
530 LSET Q$=MKI$(Q%)
540 PUT #1,PART%
550 RETURN
560 REM remove from stock
570 GOSUB 840
580 IF ASC(F$)=255 THEN PRINT "NULL ENTRY": RETURN
590 PRINT D$
600 INPUT "QUANTITY TO SUBTRACT";S%
610 Q%=CVI(Q$)
620 IF (Q%-S%)<0 THEN PRINT "ONLY";Q%;
        " IN STOCK": GOTO 600
630 Q%=Q%-S%
640 IF Q%=<CVI(R$) THEN PRINT "QUANTITY NOW";
        Q%;" REORDER LEVEL";CVI(R$)
650 LSET Q$=MKI$(Q%)
660 PUT #1,PART%
670 RETURN
680 REM list items below reorder level
690 FOR I=1 TO 100
710 GET #1,I
720 IF CVI(Q$)<CVI(R$) THEN PRINT D$;
        " QUANTITY";CVI(Q$) TAB(50)
        "REORDER LEVEL";CVI(R$)
730 NEXT I
740 RETURN
840 INPUT "PART NUMBER";PART%
850 IF (PART%<1)OR(PART%>100) THEN PRINT
        "BAD PART NUMBER": GOTO 840
        ELSE GET #1,PART%: RETURN
900 REM initialize file
910 INPUT "ARE YOU SURE";B$: IF B$<>"Y" THEN RETURN
920 LSET F$=CHR$(255)
930 FOR I=1 TO 100
```

APPENDIXES

```
940 PUT #1,I
950 NEXT I
960 RETURN
2000 REM end application
2010 CLOSE: END
```

Appendix C. Machine Language Subroutines

This appendix describes how BASIC interfaces with machine language subroutines. In particular, it describes:

- How to allocate memory for the subroutines
- How to get the machine language subroutine into memory
- How to call the subroutine from BASIC and pass parameters to it

This appendix is intended to be used by an experienced machine language programmer.

Reference Material

Rector, Russell and Alexy, George. *The 8086 Book*. Osborne/McGraw-Hill, Berkeley, California, 1980. (includes the 8088)

Intel Corporation Literature Department. *The 8086 Family User's Manual*, 9800722. 3065 Bowers Avenue, Santa Clara, CA 95051.

IBM Corporation Personal Computer library. *Macro Assembler*. Boca Raton, FL 33432.

IBM Corporation Personal Computer library. *Technical Reference*. Boca Raton, FL 33432.

APPENDIXES

Setting Memory Aside for Your Subroutines

BASIC normally uses all memory available from its starting location up to a maximum of 64K-bytes. This BASIC work area contains your BASIC program. and data, along with the interpreter work area and BASIC's stack. You may allocate memory space for machine language subroutines either inside or outside this BASIC 64K work area. Where you decide to put the routines depends on the total amount of available memory and the size of the applications to be loaded.

Your system needs more than 64K-bytes of memory if you want to put your machine language subroutines outside BASIC's 64K work area. If you are using DOS 2.00, DOS takes up about 24K-bytes, so you need 128K-byte system in order for there to be room outside the BASIC work area for the machine language subroutines.

Outside the BASIC Work Area: If your system has enough memory that you can put your subroutines outside the BASIC 64K-byte work area, you don't have to do anything to reserve that area. You use the DEF SEG statement to address the external subroutine area outside the BASIC work area.

For example, in a 128K-byte system, to specify an address beyond BASIC's workspace, you could use:

```
110 DEF SEG=&H1700
```

This statement specifies a segment starting at hexadecimal location 17000 (92K).

Remember,

DOS 2.10	=	24k-bytes
workspace	=	64k-bytes
Total	=	88k-bytes

Inside the BASIC Work area: To keep BASIC from writing over your subroutines in memory, use either:

- The CLEAR statement, which is available in all versions of BASIC

- The **/M:** option on the BASIC or BASICA command to start BASIC from DOS

Only the highest memory locations can be set aside for subroutines. For example, to reserve the highest 4K-byte area of BASIC's 64K-byte work area for your machine language subroutines, you could use:

```
10 CLEAR ,&HF000
```

or start BASIC with the DOS command:

```
BASIC /M:&HF000
```

Either of these statements restricts the size of the BASIC work area to hex F000 (60K) bytes, so you can use the uppermost 4K-bytes for machine language subroutines.

Getting the Subroutine Code into Memory

The following are offered as suggestions about how machine language subroutines can be loaded. We don't describe all possible situations.

Two common ways to get a machine language program into memory are:

- Poking it into memory from your BASIC program
- Loading it from a file on diskette or cassette

Poking a Subroutine into Memory

You can code relatively short subroutines in machine language and use the POKE statement to put the code into memory. In this way, the subroutine actually becomes a part of your BASIC program. One way to do this is:

1. Determine the machine code for your subroutine.
2. Put the hex value (&Hxx format) of each byte of the code into DATA statements.
3. Execute a loop which reads each data byte, and then pokes it into the area you've selected for the subroutine (see the preceding discussion).
4. After the loop is complete, the subroutine is loaded. If you are going to call the subroutine using the USR function, then you must execute a DEF USR statement to define the entry address of the subroutine; if you are going to call the subroutine using the CALL statement, you must set the value of the subroutine variable to the subroutine's entry address.

For example:

```
Ok
10 DEFINT A-Z
20 DEF SEG=&H1700
30 FOR I=0 TO 21
40 READ J
50 POKE I,J
60 NEXT
70 SUBRT=0
```

```
80 A=2:B=3:C=0
90 CALL SUBRT(A,B,C)
100 PRINT C
110 END
120 DATA &H55,&H8B,&HEC,&H8B,&H76,&H0A
130 DATA &H8B,&H04,&H8B,&H76,&H08
140 DATA &H03,&H04,&H8B,&H7E,&H06
150 DATA &H89,&H05,&H5D,&HCA,&H06,&H00
RUN
 5
Ok
```

Loading the Subroutine from a File

You use the BASIC BLOAD command to load a memory image file directly into memory. The memory image can be a machine language subroutine which was saved using the BSAVE command. Of course, that leads to the question of how the subroutine got there in the first place. The machine language subroutine may be an executable file which was created by the linker from DOS, and which was placed into memory using DEBUG. DEBUG and the linker are explained in the IBM Personal Computer *Disk Operating System* manual.

The following is a suggested way to use BLOAD to get such a machine language subroutine into memory:

1. Use the linker to produce an .EXE file of your routine (let's call it ASMROUT.EXE) so it will load at the HIGH end of memory.

2. Load BASIC under DEBUG by entering:

   ```
   DEBUG BASIC.COM
   ```

3. Display the registers (use the R command) to find out where BASIC was put in memory. Record the values contained in the registers (CS, IP, SS, SP, DS, ES) for later reference.

4. Use DEBUG to load the .EXE file (your subroutine) into HIGH memory, where it will overlay the transient portion of COMMAND.COM.

```
N ASMROUT.EXE
L
```

5. Display the registers (use the R command) to find out where the subroutine was placed in memory. Record the values contained in the CS and IP registers for later use.

6. Reset the registers (use the R command) back to the values they contained when BASIC.COM was originally loaded, using the values noted in step 3.

7. Pass parameters to BASIC, if required, by using the N command of DEBUG to initialize the parameter passing area.

```
N/M:40000
```

> **Note:** The /M:*max workspace* entry is required only when the subroutine is inside the BASIC 64K work area.

8. Use the G command to branch to the BASIC entry point and to set breakpoints (if desired) in the machine language subroutine.

9. When BASIC prompts, load your BASIC application program and edit the DEF SEG and either the DEF USR statement or the value of the CALL variable to correspond with the location of the subroutine as determined when you loaded the subroutine in step 5.

   - Use the previously recorded value in the CS register for DEF SEG
   - Use the previously recorded value in the IP register for the DEF USR or the variable value of the CALL

10. In direct mode in BASIC, enter a BSAVE command to save the subroutine area. Use the starting location defined by the CS and IP registers when the subroutine was loaded in step 5, and the code length printed on the assembler listing or LINK map. (Refer to "BSAVE Command" in Chapter 4.)

11. Edit your BASIC application program so it contains a BLOAD statement after the DEF SEG that sets the proper value of CS for the subroutine.

    Note: If the machine language routine is self-relocatable, BLOAD can be used to put the subroutine some place other than where the linker orginally placed it. If you make such a change, be sure to make a corresponding change to the DEF SEG statement associated with the call so that BASIC can find the subroutine at execution time.

    Some suggestions for alternate locations for the subroutine are:

    - An unused screen buffer
    - An unused file buffer (located with VARPTR(#*f*))
    - A string variable area located with VARPTR(*stringvar*)
    - A variable array area pointed to by the VARPTR(*arrayname*(0)).

    (See "BLOAD Command" and "VARPTR Function" in Chapter 4.)

12. Save the resulting changed BASIC application.

A sample BASIC routine calling assembler subroutine:

APPENDIXES

```
5  A%=2 : A%=3
10 DEF SEG=&H1700
15 BLOAD "B:ASMROUT", 0
20 SUBRT=0
30 CALL SUBRT (A%,B%,C%)
40 PRINT C%
```

Some Notes on Using DEBUG with BASIC: When you run BASIC under DEBUG, BASIC is loaded after DEBUG in memory, so DEBUG is not written over if you load a large BASIC program. If you set breakpoints in your machine language subroutine, they return you to DEBUG. The SYSTEM command also returns you from BASIC to DEBUG.

Calling the Subroutine from Your BASIC Program

All versions of BASIC have two ways to call machine language subroutines: the USR function, and the CALL statement. This section describes the use of both USR and CALL.

Common Features of CALL and USR

Whether you call your machine language subroutines with CALL or with the USR function, you must keep the following things in mind:

Entering the Subroutine

- At entry, the segment registers DS, ES, and SS are all set to the same value, the address of BASIC's data space (the default for DEF SEG).
- At entry, the code segment register, CS, contains the current value specified in the latest DEF SEG. If DEF SEG has not been specified, or if the latest DEF SEG did not specify an override value, the value in CS is the same as in the other three segment registers.

String Arguments

- If an input argument is a string, the value received in the argument is the address of a three-byte area called the *string descriptor*:

  1. Byte 0 of the string descriptor contains the length of the string (0 to 255).

  2. Byte 1 of the string descriptor contains the lower 8 bits of the offset of the string in BASIC's data space.

APPENDIXES

3. Byte 2 of the string descriptor contains the higher 8 bits of the offset of the string in BASIC's data space.

The string itself is pointed to by the last two bytes of the string descriptor.

Warning:
The subroutine must not change the contents of any of the three bytes of the ***string descriptor***.

The subroutine may change the *content* of the string itself, but not its *length.*

If the subroutine changes a string, be aware that this may *modify your program*. The following example may change the string "TEXT" in the BASIC program.

```
A$ = "TEXT"
CALL SUBRT(A$)
```

However, the next example does not change the program, because the string concatenation causes BASIC to copy the string into the string space where it may be safely changed without affecting the original text.

```
A$ = "BASIC"+""
CALL SUBRT(A$)
```

Returning from the Subroutine

- The return to BASIC must be by an inter-segment RET instruction. (The subroutine is a FAR procedure.)

- At exit, all segment registers and the stack pointer, SP, must be restored. All other registers (and flags) may be altered.

- The stack pointer, at entry, indicates a stack that has only 16 bytes (eight words) available for use by

the subroutine. If more stack space is needed, the subroutine must set up its own stack segment and stack pointer. You should make sure that the location of the current stack is recorded so its pointer can be restored just before return.

- If interrupts were disabled by the subroutine, they should be enabled before return.

CALL Statement

Machine language subroutines may be called using the BASIC CALL statement. The format of the CALL statement is:

CALL *numvar* [(*variable list*)]

numvar is the name of a numeric variable. Its value is the offset, from the segment set by DEF SEG, that is the starting point in memory of the subroutine being called.

variable list
contains the variables, separated by commas, that are to be passed as arguments to the routine. (The arguments cannot be constants.)

Execution of a CALL statement causes the following:

1. For each variable in the variable list, the variable's location is pushed onto the stack. The location is specified as a two-byte offset into BASIC's data segment (the default DEF SEG).

2. The return address specified in the CS register and the offset are pushed onto the stack.

3. Control is transferred to the machine language routine using the segment address specified in the last DEF SEG statement and the offset specified by the value of *numvar*.

Notes for the CALL Statement

- You can return values to BASIC through the arguments by changing the values of the variables in the argument list.
- If the argument is a string, the offset for the argument points to the three-byte *string descriptor* as explained previously.
- The called routine must know how many arguments were passed. Parameters are referenced by adding a positive offset to BP after the called routine moves the current stack pointer into BP. The first instructions in the subroutine should be:

```
PUSH BP      ;SAVE BP
MOV BP,SP    ;MOVE SP TO BP
```

The offset into the stack of any one particular argument is calculated as follows:
offset from BP = $2*(n-m)+6$

where:

n is the total number of arguments passed.

m is the position of the specific argument in the argument list of the BASIC CALL statement (*m* may range from 1 to *n*).

Example: The following example adds the values in A% and B% and stores the result in C%:

The following statements are in BASIC:

```
100 A%=2: B%=3
200  DEF SEG=&H1700
250  BLOAD "SUBRT.EXE",0
300  SUBRT=0
400  CALL SUBRT (A%,B%,C%)
500 PRINT C%
```

> **Note:** Line 200 sets the segment to location hex 17000. SUBRT is set to 0 so that the call to SUBRT executes the subroutine at location &H17000.

The following statements are in IBM Personal Computer Macro Assembler source code:

```
CSEG    SEGMENT
        ASSUME    CS:CSEG
SUBRT   PROC      FAR
        PUSH BP              ;SAVE BP
        MOV BP,SP            ;SET BASE PARM LIST
        MOV SI,[BP]+10       ;GET ADDR PARM A
        MOV AX,[SI]          ;GET VALUE OF A
        MOV SI,[BP]+8        ;GET ADDR PARM B
        ADD AX,[SI]          ;ADD VALUE B TO REG
        MOV DI,[BP]+6        ;GET ADDR PARM C
        MOV [DI],AX          ;PASS BACK SUM
        POP BP               ;RESTORE BP
        RET 6                ;FAR RETURN TO BASIC
SUBRT   ENDP
CSEG    ENDS
        END
```

> **Note:** When you call a routine using the CALL statement, the routine must return with a RET *n*, where *n* is 2 times the number of arguments in the variable list. This is necessary to adjust the stack to the point at the start of the calling sequence.

As another example:

```
10 DEFINT A-Z
100 DEF SEG=&H1800
110 BLOAD "SUBRT.EXE",0
120 SUBRT=0
130 CALL SUBRT (A,B$,C)
```

The following sequence of Macro Assembler code shows how the arguments (including the address of a string descriptor) are passed and accessed, and how the result is stored in variable C:

```
PUSH  BP          ;SAVE BP
MOV   BP,SP       ;GET CURRENT STK POSITION INTO BP
MOV   BX,[BP]+8   ;GET ADDR OF B$ STRING DESCRIPTOR
MOV   CL,[BX]     ;GET LENGTH OF B$ INTO CL
MOV   DX,1[BX]    ;GET ADDR OF B$ TEXT INTO DX
 .
 .
 .
MOV   SI,[BP]+10  ;GET ADDR OF A INTO SI
MOV   DI,[BP]+6   ;GET ADDR OF C INTO DI
MOVS  WORD        ;STORE VARIABLE A INTO C
POP   BP          ;RESTORE BP
RET   6           ;RESTORE STACK, RETURN
END
```

> **Warning:** It is entirely up to you to make sure that the arguments in the CALL statement match in number, type, and length with the arguments expected by the subroutine.

In the preceding example, the instruction **MOVS WORD** copies only two bytes because variables A and C are integers. However, if A and C are single-precision, four bytes must be copied; if A and C are double-precision, eight bytes must be copied.

USR Function Calls

The other way to call machine language subroutines from BASIC is with the USR function. The format of the USR function is:
USR[*n*](*arg*)

n must be a single digit in the range 0 through 9.

arg is any numeric expression or a string variable name.

n specifies which USR routine is being called, and corresponds with the digit supplied in the DEF USR statement for that routine. If *n* is omitted, USR0 is assumed. The address specified in the DEF USR statement determines the starting address of the subroutine. Even if the subroutine does not require an argument, a dummy argument must still be supplied.

When the USR function is called, register AL contains a value that specifies the type of argument that was supplied. The value in AL will be one of the following:

Value in AL	Type of Argument
2	Two-byte integer (two's complement)
3	String
4	Single-precision number
8	Double-precision number

If the argument is a string, the DX register points to the three-byte string descriptor. (See "Common Features of CALL and USR," described previously.)

If the argument is a number and not a string, the value of the argument is placed in the Floating Point Accumulator (FAC), which is an eight-byte area in

APPENDIXES

BASIC's data space. In this case, the BX register contains the offset within the BASIC data space to the fifth byte of the eight-byte FAC. For the following examples, assume that the FAC is in bytes hex 49F through hex 4A6; that is, BX contains hex 4A3:

If the argument is an integer:

- Hex 4A4 contains the upper 8 bits of the argument.

- Hex 4A3 contains the lower 8 bits of the argument.

If the argument is a single-precision number:

- Hex 4A6 contains the exponent minus 128, and the binary point is to the left of the most significant bit of the mantissa. Hex 4A5 contains the highest 7 bits of the mantissa with the leading 1 suppressed (implied). Bit 7 is the sign of the number (0 = positive; 1 = negative).

- Hex 4A4 contains the middle 8 bits of the mantissa.

- Hex 4A3 contains the lowest 8 bits of the mantissa.

If the argument is a double-precision number:

- Hex 4A3 through hex 4A6 are the same as described under single-precision floating-point number in the preceding paragraph.

- Hex 49F through Hex 4A2 contain four more bytes of the mantissa (hex 49F contains the lowest 8 bits).

Usually, the value returned by a USR function is the same type (integer, string, single-precision, or double-precision) as the argument that was passed to it. However, a numerical argument of the function, regardless of its type, may be forced to an integer value by calling the FRCINT routine to get the integer equivalent of the argument placed into register BX.

If the value being returned by the function is to be an integer, place the resulting value into the BX register. Then make a call to MAKINT just before the inter-segment return. This passes the value back to BASIC by placing it into the FAC.

The methods for accessing FRCINT and MAKINT are shown in the following example:

```
100 DEF SEG=&H1800
120 BLOAD "SUBRT.EXE",0
130 DEF USR0=0
140 X = 5 'Note that X is single-precision
150 Y = USR0(X)
160 PRINT Y
```

At location 1800:0 (segment:offset), the following Macro Assembler language routine has been loaded. The routine doubles the argument passed and returns an integer result:

```
RSEG        SEGMENT AT 0F600H ;BASE OF BASIC ROM
            ORG    3       ;OFFSET TO FORCE INTEGER
  FRCINT    LABEL FAR
            ORG    7       ;OFFSET TO MAKE INTEGER
  MAKINT    LABEL FAR
RSEG       ENDS
.
.
.
CSEG        SEGMENT
USRPRG      PROC   FAR     ;ENTRY POINT
            CALL   FRCINT  ;FORCE ARG IN FAC INTO [BX]
            ADD    BX,BX   ;[BX] = [BX] * 2
            CALL   MAKINT  ;PUT INT RSLT IN BX INTO FAC
            RET            ;INTER-SEGMENT RETURN TO BASIC
USRPRG      ENDP
CSEG        ENDS
```

Note: FRCINT and MAKINT perform inter-segment returns. You should make sure that the calls to FRCINT and MAKINT are defined by a FAR procedure.

Notes

Appendix D. Converting Programs to PC jr BASIC

Since PCjr BASIC is similar to many microcomputer BASICs, the PCjr will support programs written for a wide variety of microcomputers. If you have programs written in a BASIC other than the PCjr BASIC, some minor adjustments may be necessary before running them with PCjr BASIC. Here are some specific things to look for when converting BASIC programs.

File I/O

In PCjr BASIC, you read and write information to a file on diskette or cassette by opening the file to associate it with a particular file number; then by using particular I/O statements which specify that file number. I/O to diskette and cassette files is implemented differently in some other BASICs. Refer to the section in Chapter 3 called "Files," and to "OPEN Statement" in Chapter 4 for more specific information.

Also, in PCjr BASIC, random file records are automatically blocked as appropriate to fit as many records as possible in each sector.

FOR–NEXT Loops

In PCjr BASIC, when you exit from a FOR–NEXT loop, the counter is set to the first unused value. This differs from some other BASICs where the counter is set to the last value used. For example:

```
10 FOR I=1 to 5
20 NEXT
30 PRINT I
```

The result on the PCjr is 6; in another BASIC it may be 5.

Graphics

How you draw on the screen varies greatly between different BASICs. Refer to the discussion of graphics in Chapter 3 for specific information about PCjr BASIC.

IF–THEN

The IF statement in PCjr BASIC contains an optional ELSE clause, which is performed when the expression being tested is false. Some other BASICs do not have this capability. For example, in another BASIC you may have:

```
10 IF A=B THEN 30
20 PRINT "NOT EQUAL" : GOTO 40
30 PRINT "EQUAL"
40 REM CONTINUE
```

This sequence of code will still function correctly in PCjr BASIC, but it may also be conveniently recoded as:

```
10 IF A=B THEN PRINT "EQUAL" ELSE PRINT "NOT EQUAL"
20 REM CONTINUE
```

PCjr BASIC also allows multiple statements in both the THEN and ELSE clauses. This may cause a program written in another BASIC to perform differently. For example:

```
10 IF A=B THEN GOTO 100 : PRINT "NOT EQUAL"
20 REM CONTINUE
```

In some other BASICs, if the test A=B is false, control branches to the next *statement*; that is, if A is not equal to B, "NOT EQUAL" is printed. In PCjr BASIC, both GOTO 100 and PRINT "NOT EQUAL" are

considered to be part of the THEN clause of the IF statement. If the test is false, control continues with the next program *line*; that is, to line 20 in this example. PRINT "NOT EQUAL" can never be executed.

This example can be recoded in PCjr BASIC as:

```
10 IF A=B THEN GOTO 100 ELSE PRINT "NOT EQUAL"
20 REM CONTINUE
```

Line Feeds

In other BASICs, when you enter a line feed, a line feed character is actually inserted into the text. On the PCjr, entering a line feed will pad the rest of the display line with spaces – it does not insert the line feed character. If you try to load a program with line feed characters in it, you will get a **Direct statement in file** error.

Logical Operations

In PCjr BASIC, logical operations (NOT, AND, OR, XOR, IMP, and EQV) are performed bit-by-bit on integer operands to produce an integer result. In some other BASICs, the operands are considered to be simple "true" (non-zero) or "false" (zero) values, and the result of the operation is either true or false. As an example of this difference, consider this small program:

```
10 A=9: B=2
20 IF A AND B THEN PRINT "BOTH A AND B ARE TRUE"
```

This example in another BASIC will perform as follows: A is non-zero, so it is true; B is also non-zero, so it is also true; because both A and B are true, A AND B is true, so the program prints **BOTH A AND B ARE TRUE**.

However, PCjr BASIC calculates it differently: A is 1001 in binary form, and B is 0010 in binary form, so

A AND B (calculated bit-by-bit) is 0000, or zero; zero indicates false, so the message is *not* printed, and the program continues with the next line.

This can affect not only tests made in IF statements, but calculations as well. To get similar results, recode logical expressions like the following:

```
10 A=9: B=2
20 IF (A<>0) AND (B<>0)
     THEN PRINT "BOTH A AND B ARE TRUE"
```

MAT Functions

Programs using the MAT functions available in some BASICs must be rewritten using FOR–NEXT loops to execute properly.

Multiple Assignments

Some BASICs allow statements of the form:

```
10 LET B=C=0
```

to set B and C equal to zero. PCjr BASIC would interpret the second equal sign as a logical operator and set B equal to -1 if C equaled 0. Instead, convert this statement to two assignment statements:

```
10 C=0:B=0
```

Multiple Statements

Some BASICs use a backslash (&intdiv.) to separate multiple statements on a line. With PCjr BASIC, be sure all statements on a line are separated by a colon (:).

PEEKs and POKEs

Many PEEKs and POKEs are dependent on the particular computer you are using. You should examine the *purpose* of the PEEKs and POKEs in a program in another BASIC, and translate the statement so it performs the same function on the PCjr.

Relational Expressions

In PCjr BASIC, the value returned by a relational expression, such as A>B, is either -1, indicating the relation is true, or 0, indicating the relation is false. Some other BASICs return a positive 1 to indicate true. If you use the value of a relational expression in an arithmetic calculation, the results are likely to be different from what you want.

Remarks

Some BASICs allow you to add remarks to the end of a line using the exclamation point (!). Be sure to change this to a single quote (') when converting to PCjr BASIC.

Rounding of Numbers

PCjr BASIC rounds single- or double-precision numbers when it requires an integer value. Many other BASICs truncate instead. This can change the way your program runs, because it affects not only assignment statements (for example, I%=2.5 results in I% equal to 3), but also affects function and statement evaluations (for example, TAB(4.5) goes to the fifth position, A(1.5) is the same as A(2), and X=11.5 MOD 4 will result in a value of 0 for X). Note in particular that rounding may cause PCjr BASIC to select a different element from an array than another BASIC — possibly one that is out of range!

Sounding the Bell

Some BASICs require PRINT CHR$(7) to send an ASCII bell character. In PCjr BASIC, you may replace this statement with BEEP, although it is not required.

String Handling

String Length: Since strings in PCjr BASIC are all variable length, you should delete all statements that are used to declare the length of strings. A statement such as DIM A$(I,J), which dimensions a string array for J elements of length I, should be converted to the PCjr BASIC statement DIM A$(J).

Concatenation: Some BASICs use a comma or ampersand for string concatenation. Each of these must be changed to a plus sign, which is the operator for PCjr BASIC string concatenation.

Substrings: In PCjr BASIC, the MID$, RIGHT$, and LEFT$ functions are used to take substrings of strings. Forms such as A$(I) to access the Ith character in A$, or A$(I,J) to take a substring of A$ from position I to position J, must be changed as follows:

Other BASIC	PCjr BASIC
X$=A$(I)	X$=MID$(A$,I,1)
X$=A$(I,J)	X$=MID$(A$,I,J-I+1)

If the substring reference is on the left side of an assignment and X$ is used to replace characters in A$, convert as follows:

Other BASIC	PCjr BASIC
A$(I)=X$	MID$(A$,I,1)=X$
A$(I,J)=X$	MID$(A$,I,J-I+1)=X$

Use of Blanks

Some BASICs allow statements with no separation of keywords:

```
20FORI=1TOX
```

With PCjr BASIC be sure all keywords are separated by a space:

```
20 FOR I=1 TO X
```

Other

The BASIC language on another computer may be different from the PCjr BASIC in other ways than those listed here. You should become familiar with PCjr BASIC as much as possible to be able to appropriately convert any function you may require.

Notes

Appendix F. Communications

This appendix describes the BASIC statements required to support RS232 asynchronous communication with other computers and peripherals.

Opening a Communications File

OPEN "COM... allocates a buffer for I/O in the same fashion as OPEN for diskette files. Refer to "OPEN "COM... Statement" in Chapter 4.

Communication I/O

Since each communications port is opened as a file, all input/output statements that are valid for diskette files are valid for communications.

Communications sequential input statements are the same as those for diskette files. They are:

INPUT #
LINE INPUT #
INPUT$

Communications sequential output statements are the same as those for diskette files, and are:

PRINT #
PRINT # USING
WRITE #

Refer to the INPUT and PRINT sections for details of coding syntax and usage.

GET and PUT for Communications Files

GET and PUT are only slightly different for communications files than for diskette files. They are used for fixed length I/O from or to the communications file. In place of specifying the record number to be read or written, you specify the number of bytes to be transferred into or out of the file buffer. This number cannot exceed the value set by the LEN option on the OPEN "COM... statement. Refer to the GET and PUT sections in Chapter 4.

I/O Functions

The most difficult aspect of asynchronous communication is being able to process characters as fast as they are received. At rates of 1200 bps or higher, it may be necessary to suspend character transmission from the other computer long enough to "catch up." This can be done by sending XOFF (CHR$(19)) to the other computer and XON (CHR$(17)) when ready to resume. XOFF tells the other computer to stop sending, and XON tells it it can start sending again.

> **Note:** This is a commonly used convention, but it is not universal. It depends on the protocol implemented between you and the other computer or peripheral.

Cartridge BASIC provides three functions which help in determining when an "overrun" condition may occur. These are:

LOC(f) Returns the number of characters in the input buffer waiting to be read. If the number is greater than 255, LOC returns 255.

LOF(f) Returns the amount of free space in the input buffer. This is the same as

	n-LOC(f), where *n* is the size of the communications buffer as set by the **/C:** option on the BASIC command. The default for *n* is 256.
EOF(f)	Returns true (-1) if the input buffer is empty; false (0) if there are any characters waiting to be read.

Note: A **Communication buffer overflow** can occur if a read is attempted after the input buffer is full (that is, when LOF(f) returns 0).

INPUT$ Function

The INPUT$ function is preferred over the INPUT # and LINE INPUT # statements when reading communications files, since all ASCII characters may be significant in communications. INPUT # is least desirable because input stops when a comma (,) or carriage return is seen. LINE INPUT # stops when a carriage return is seen.

INPUT$ allows all characters read to be assigned to a string. INPUT$(*n,f*) will return *n* characters from the #*f* file. The following statements are efficient for reading a communications file:

```
110 WHILE NOT EOF(1)
120 A$=INPUT$(LOC(1),#1)
.
.
.
(process data returned in A$)
.
.
.
190 WEND
```

These statements return the characters in the buffer into A$ and process them, as long as there are characters in the input buffer. If there are more than 255 characters in the buffer, only 255 will be returned

APPENDIXES

at a time to prevent **String overflow**. Further, if this is the case, EOF(1) is false and input continues until the input buffer is empty. Simple, concise, and fast.

To process characters quickly, avoid, if possible, examining every character as you receive it. If you are looking for special characters (such as control characters), you can use the INSTR function to find them in the input string.

Sample Program 1

The following program allows the PCjr to be used as a conventional "dumb" terminal in a full duplex mode. This program assumes a 300 bps line and an input buffer of 256 bytes (the **/C:** option was not used on the BASIC command).

```
10 REM   dumb terminal example
20 'set screen to black and white text mode
30 '    and set width to 40
40 SCREEN 0,0: WIDTH 40
50 'turn off soft key display; clear screen;
60 '   make sure all files are closed
70 KEY OFF: CLS: CLOSE
80 'define all numeric variables as integer
90 DEFINT A-Z
100 'define true and false
110 FALSE=0: TRUE= NOT FALSE
120 'define the XON, XOFF characters
130 XOFF$=CHR$(19): XON$=CHR$(17)
140 'open communications to file number 1,
150 ' 300 bps, EVEN parity, 7 data bits
160 OPEN "COM1:300,E,7" AS #1
170 'use screen as a file, just for fun
180 OPEN "SCRN:" FOR OUTPUT AS 2
190 'turn cursor on
200 LOCATE ,,1
400 PAUSE=FALSE: ON ERROR GOTO 9000
490 '
500 'send keyboard input to com line
510 B$=INKEY$: IF B$<>"" THEN PRINT #1,B$;
520 'if no chars in com buffer, check key in
530 IF EOF(1) THEN 510
540 'if buffer more than 1/2 full, then
550 '   set PAUSE flag to say input suspended,
560 '   send XOFF to host to stop transmission
```

```
570 IF LOC(1)>128 THEN PAUSE=TRUE: PRINT #1,XOFF$;
580 'read contents of com buffer
590 A$=INPUT$(LOC(1),#1)
600 'get rid of linefeeds to avoid double spaces
610 '    when input displayed on screen
620 LFP=0
630 LFP=INSTR(LFP+1,A$,CHR$(10)) 'look for LF
640 IF LFP>0 THEN MID$(A$,LFP,1)=" ": GOTO 630
650 'display com input, and check for more
660 PRINT #2,A$;: IF LOC(1)>0 THEN 570
670 'if transmission suspended by XOFF,
680 '    resume by sending XON
690 IF PAUSE THEN PAUSE=FALSE: PRINT #1,XON$;
700 'check for keyboard input again
710 GOTO 510
8999 'if error, display error number and retry
9000 PRINT "ERROR NO.";ERR: RESUME
```

Notes on the Program

- "Asynchronous" communication implies character I/O as opposed to line or block I/O. Therefore, all PRINTs (either to communications file or to screen) are ended with a semicolon (;). This stops the carriage return normally issued at the end of the list of values to be printed.

- Line 90, where all numeric variables are defined as integer, is coded because any program looking for speed optimization should use integer counters in loops where possible.

- Note in line 510 that INKEY$ will return a null string if no character is pending.

- Note: Commands entered into the IBM Internal Modem will not be echoed back to the program. Therefore, if you desire to see command data as being entered, your program must display it to the screen. Command data can be interpreted as all input between CTL n and Carriage Return.

APPENDIXES

Sample Program 2

Characters received by the communiction port during the time that the diskette is being written to are lost. This presents a special problem for the BASIC program that writes communication data to a disk file.

Therefore, we must suspend data transmission from the other computer long enough to write data to the diskette.

The following BASIC program does this by sending XOFF to the other computer just before writing to the disk file. After the data is written, an XON character is sent to the host telling it to resume transmission.

> **Note:** This is the only way to reliably write data to disk from the communication port. If the computer you are connected to does not have a means for suspending transmission, then characters will invariably be lost....

This program assumes that data is being sent to it from another computer. Data received is printed on the screen while being written to a disk file called DATA.LOG. The program stops, closes all files, and returns to BASIC's direct mode when a Ctrl-Z character (CHR$(26)) is received:

```
1  XOFF$=CHR$(19) : XON$=CHR$(17)
10 OPEN "COM1:300,E,7" AS 1
20 OPEN "SCRN:" FOR OUTPUT AS 2
30 OPEN "DISK.DAT" FOR OUTPUT AS 3
40 ON COM(1) GOSUB 2000:COM(1) ON 'If buffer gets full, send XOFF
50 IF EOF(1) THEN 50  'Wait until something to read
60 A$=INPUT$(1,1)  'read 1 character from comm
70 IF A$=CHR$(26) THEN 900   'Go close files if EOF
80 PRINT A$:  'Echo character to screen
90 GOSUB 1000 'Suspend transmission, write to disk
100 GOTO 50   'Repeat until EOF
110 '
900 CLOSE:END
999 '
1000 IF PAUSE THEN 1020 'Don't suspend if already stopped
1010 PRINT #1,XOFF$;:PAUSE= -1
```

```
1020 FOR I=1 TO 400:NEXT I 'Some time to be sure stopped
1030 PRINT #3,A$;  'Write to disk file
1050 IF LOC(1)<16 AND PAUSE THEN PRINT #1,XON$;:PAUSE=0
1060 RETURN
1070 '
2000 IF LOC(1) > 224 THEN PRINT #1,XOFF$;:PAUSE= -1
2010 RETURN
```

Operation of Control Signals

This section contains more detailed technical information that you may need to know to communicate with another computer or peripheral from BASIC.

The output from the Asynchronous Communications Port conforms to the EIA RS232-C standard for interface between Data Terminal Equipment (DTE) and Data Communications Equipment (DCE). This standard defines several control signals that are transmitted or received by your PCjr to control the interchange of data with another computer or peripheral. These signals are DC voltages that are either ON (greater than +3 volts) or OFF (less than -3 volts). See the IBM Personal Computer *Technical Reference* manual for details.

Control of Output Signals with OPEN

When you start BASIC on your PCjr, the RTS (Request To Send) and DTR (Data Terminal Ready) lines are held OFF. When an OPEN "COM... statement is performed, both of these lines are normally turned ON. However, you can specify the **RS** option on the OPEN "COM... statement to suppress the RTS signal. The lines remain ON until the communications file is closed (by CLOSE, END, NEW, RESET, SYSTEM, or RUN without the **R** option). Even if the OPEN "COM... statement fails with an error (as described below), the DTR line (and RTS line, if

APPENDIXES

applicable) is turned ON and stays ON. This allows you to retry the OPEN without having to execute a CLOSE.

Use of Input Control Signals

Normally, if either the CTS (Clear To Send) or DSR (Data Set Ready) lines are OFF, then an OPEN "COM... statement will not execute. After one second, BASIC will return with a **Device Timeout** error (error code 24). The Carrier Detect (sometimes called Receive Line Signal Detect) can be either ON or OFF; it has no effect on the operation of the program.

However, you can specify how you want these lines tested with the **RS, CS, DS,** and **CD** options on the OPEN "COM... statement. Refer to "OPEN "COM... Statement" in Chapter 4 for details.

If any of the signals that are being tested are turned OFF while the program is executing, I/O statements associated with the communications file won't work. For example, when you execute a PRINT # statement after the CTS or DSR line is turned off, a **Device Fault** (code 25) or **Device Timeout** (code 24) error occurs. The RTS and DTR remain on even if such an error occurs.

You can test for a line disconnect by using the INP function to read the bits in the MODEM Status Register on the Asynchronous Communications Port. See the following section, "Testing for Modem Control Signals," for details.

Testing for Modem Control Signals

There are four input control signals picked up by the Asynchronous Communications Port. These signals are the CTS and DSR signals described previously, the Carrier Detect (sometimes called Received Line Signal

Detect) (pin 8), and Ring Indicator (pin 22). You can specify how you want to test the CTS, DSR, and CD lines with the OPEN "COM... statement. Ring Indicator is not used at all by the communications function in BASIC.

If you need to test for any of these signals in a program, you can check the bits corresponding to these signals in the MODEM Status Register on the Asynchronous Communications Port. To read the eight bits in this register, you use the INP function—use INP(&H3FE) to read the register on the IBM internal modem, and INP(&H2FE) to read the register on the RS232 Serial Port. See the "Asynchronous Communications Port" section of the PCjr *Technical Reference* manual for a description of which bits in the Status Register correspond to which control signals. You can also use the Delta bits in this register to determine if transient signals have appeared on any of the control lines. Note that for a control signal to have meaning, the pin corresponding to that signal must be connected in the cable to your modem or to the other computer.

You can also test for bits in the Line Status Register on the Asynchronous Communications Port. Use INP(&H3FD) to access this register on the IBM Internal Modem, and INP(&H2FD) to access it on a RS232 Serial Port. Again, the bits are described in the PCjr *Technical Reference* manual. These bits can be used to determine what types of errors have occurred on receipt of characters from the communications line or whether a break signal has been detected.

APPENDIXES

Direct Control of Output Control Signals

You can control the RTS or DTR control signals directly from a BASIC program with an OUT statement. The states (ON or OFF) of these signals are controlled by bits in the MODEM Control Register on the Asynchronous Communications Port. The address of this register is &H3FC on the IBM Internal Modem

and &H2FC on the RS232 Serial Port. The PCjr *Technical Reference* manual describes which of these bits correspond to which signals.

You can also change bits in the Line Control Register on the Asynchronous Communications Port. You should be careful in changing these bits as most of the bits in this register have been set by BASIC at the time an OPEN statement is executed and changing a bit could cause communications failure. The Line Control Register is at address &H3FB on the IBM Internal Modem and at address &H2FB on the RS232 Serial Port.

When changing bits in either the MODEM Control Register or the Line Control Register, you should first read the register (with an INP function) and then rewrite the register with only the pertinent bit or bits changed.

A bit you may wish to control in the Line Control Register is bit 6, the Set Break bit. This bit permits you to produce a Break signal on the communications send line. A Break is often used to signal a remote computer to stop transmission. Typically a Break lasts for half a second. To produce such a signal, you must turn ON the Set Break, wait for the desired time of the Break signal, and then turn the bit OFF. The following BASIC statements will produce a Break signal of about half a second duration on the IBM Internal Modem.

```
100 IC%=INP(&H3FB) 'get contents of modem register
110 IZ%=IC% OR &H40 'turn ON the Set Break bit
110 OUT &H3FB,IZ%  'transmit to modem control register
120 FOR I=1 TO 500: NEXT I 'delay half a second
130 OUT &H3FB,IC% 'turn Set Break bit OFF in register
```

Communication Errors

Errors occur on communication files in the following order:

1. When opening the file—

    a. **Device Timeout** if one of the signals to be tested (CTS, DSR, or CD) is missing.

2. When reading data—

    a. **Com buffer overflow** if overrun occurs.

    b. **Device I/O error** for overrun, break, parity, or framing errors.

    c. **Device Fault** if you lose DSR or CD.

3. When writing data—

    a. **Device Fault** if you lose CTS, DSR, or CD on a Modem Status Interrupt while BASIC was doing something else.

    b. **Device Timeout** if you lose CTS, DSR, or CD while waiting to put data in the output buffer.

Notes

Appendix G. ASCII Character Codes

The following table lists all the ASCII codes (in decimal) and their associated characters. These characters can be displayed using PRINT CHR$(*n*), where *n* is the ASCII code. The column headed "Control Character" lists the standard interpretations of ASCII codes 0 to 31 (usually used for control functions or communications).

Each of these characters may be entered from the keyboard by pressing Alt-Fn/n, then pressing and holding the Alt key, then pressing the digits for the ASCII code. Note, however, that some of the codes have special meaning to the BASIC program editor—the program editor uses its own interpretation for the codes and may not display the special character listed here. To reset, press and hold the Alt key and then the Fn/n key.

ASCII value	Character	Control character	ASCII value	Character
000	(null)	NUL	032	(space)
001	☺	SOH	033	!
002	☻	STX	034	"
003	♥	ETX	035	#
004	♦	EOT	036	$
005	♣	ENQ	037	%
006	♠	ACK	038	&
007	(beep)	BEL	039	'
008	◘	BS	040	(
009	(tab)	HT	041	)
010	(line feed)	LF	042	*
011	(home)	VT	043	+
012	(form feed)	FF	044	,
013	(carriage return)	CR	045	-
014	♫	SO	046	.
015	☼	SI	047	/
016	►	DLE	048	0
017	◄	DC1	049	1
018	↕	DC2	050	2
019	‼	DC3	051	3
020	¶	DC4	052	4
021	§	NAK	053	5
022	▬	SYN	054	6
023	↨	ETB	055	7
024	↑	CAN	056	8
025	↓	EM	057	9
026	→	SUB	058	:
027	←	ESC	059	;
028	(cursor right)	FS	060	<
029	(cursor left)	GS	061	=
030	(cursor up)	RS	062	>
031	(cursor down)	US	063	?

ASCII value	Character
064	@
065	A
066	B
067	C
068	D
069	E
070	F
071	G
072	H
073	I
074	J
075	K
076	L
077	M
078	N
079	O
080	P
081	Q
082	R
083	S
084	T
085	U
086	V
087	W
088	X
089	Y
090	Z
091	[
092	\
093	]
094	^
095	_

ASCII value	Character
096	`
097	a
098	b
099	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r
115	s
116	t
117	u
118	v
119	w
120	x
121	y
122	z
123	{
124	¦
125	}
126	~
127	⌂

ASCII value	Character
128	Ç
129	ü
130	é
131	â
132	ä
133	à
134	å
135	ç
136	ê
137	ë
138	è
139	ï
140	î
141	ì
142	Ä
143	Å
144	É
145	æ
146	Æ
147	ô
148	ö
149	ò
150	û
151	ù
152	ÿ
153	Ö
154	Ü
155	¢
156	£
157	¥
158	Pt
159	ƒ

ASCII value	Character
160	á
161	í
162	ó
163	ú
164	ñ
165	Ñ
166	ª
167	º
168	¿
169	⌐
170	¬
171	½
172	¼
173	¡
174	«
175	»
176	░
177	▒
178	▓
179	│
180	┤
181	╡
182	╢
183	╖
184	╕
185	╣
186	║
187	╗
188	╝
189	╜
190	╛
191	┐

PRINT CHR$(

ASCII value	Character	ASCII value	Character
192	└	224	α
193	┴	225	β
194	┬	226	Γ
195	├	227	π
196	─	228	Σ
197	┼	229	σ
198	╞	230	μ
199	╟	231	τ
200	╚	232	Φ
201	╔	233	Θ
202	╩	234	Ω
203	╦	235	δ
204	╠	236	∞
205	═	237	Ø
206	╬	238	ϵ
207	╧	239	∩
208	╨	240	≡
209	╤	241	±
210	╥	242	≥
211	╙	243	≤
212	╘	244	⌠
213	╒	245	⌡
214	╓	246	÷
215	╫	247	≈
216	╪	248	°
217	┘	249	•
218	┌	250	·
219	█	251	√
220	▄	252	^{n}
221	▌	253	^{2}
222	▐	254	■
223	▀	255	(blank 'FF')

APPENDIXES

Extended Codes

For certain keys or key combinations that cannot be represented in standard ASCII code, an extended code is returned by the INKEY$ system variable. A null character (ASCII code 000) will be returned as the first character of a two-character string. If a two-character string is received by INKEY$, then you should go back and examine the second character to determine the actual key pressed. Usually, but not always, this second code is the scan code of the primary key that was pressed. The ASCII codes (in decimal) for this second character, and the associated key(s) are listed below.

Second Code	Meaning
3	(null character) NUL
15	(shift tab) |<--
16-25	Alt- Q, W, E, R, T, Y, U, I, O, P
30-38	Alt- A, S, D, F, G, H, J, K, L
44-50	Alt- Z, X, C, V, B, N, M
59-68	function keys Fn plus F1 through F10 (when disabled as soft keys)
71	Fn/Home
72	Home
73	Fn/Pg Up
75	Pg Up
77	Pg Dn
79	Fn/End
80	End
81	Fn/Pg Dn
82	Ins
83	Del
84-93	F11-F20 (Shift- Fn/F1 through F10)
94-103	F21-F30 (Ctrl- Fn/F1 through F10)
104-113	F31-F40 (Alt- Fn/F1 through F10)
114	Fn/Prtsc
115	Ctrl-Pg Up (Previous Word)
116	Ctrl-Pg Dn (Next Word)
117	Ctrl-Fn/End
118	Ctrl-Fn/Pg Dn

119	Ctrl-Fn/Home
120-131	Alt- 1,2,3,4,5,6,7,8,9,0,-,=
132	Ctrl-Fn/Pg Up

Notes

Appendix H. Hexadecimal Conversion Tables

Hex	Decimal	Hex	Decimal
1	1	10	16
2	2	20	32
3	3	30	48
4	4	40	64
5	5	50	80
6	6	60	96
7	7	70	112
8	8	80	128
9	9	90	144
A	10	A0	160
B	11	B0	176
C	12	C0	192
D	13	D0	208
E	14	E0	224
F	15	F0	240
100	256	1000	4096
200	512	2000	8192
300	768	3000	12288
400	1024	4000	16384
500	1280	5000	20480
600	1536	6000	24576
700	1792	7000	28672
800	2048	8000	32768
900	2304	9000	36864
A00	2560	A000	40960
B00	2816	B000	45056
C00	3072	C000	49152
D00	3328	D000	53248
E00	3584	E000	57344
F00	3840	F000	61440

Binary to Hexadecimal Conversion Table

Binary bit pattern	Hex value	Decimal value
0001	1	1
0010	2	2
0011	3	3
0100	4	4
0101	5	5
0110	6	6
0111	7	7
1000	8	8
1001	9	9
1010	A	10
1011	B	11
1100	C	12
1101	D	13
1110	E	14
1111	F	15

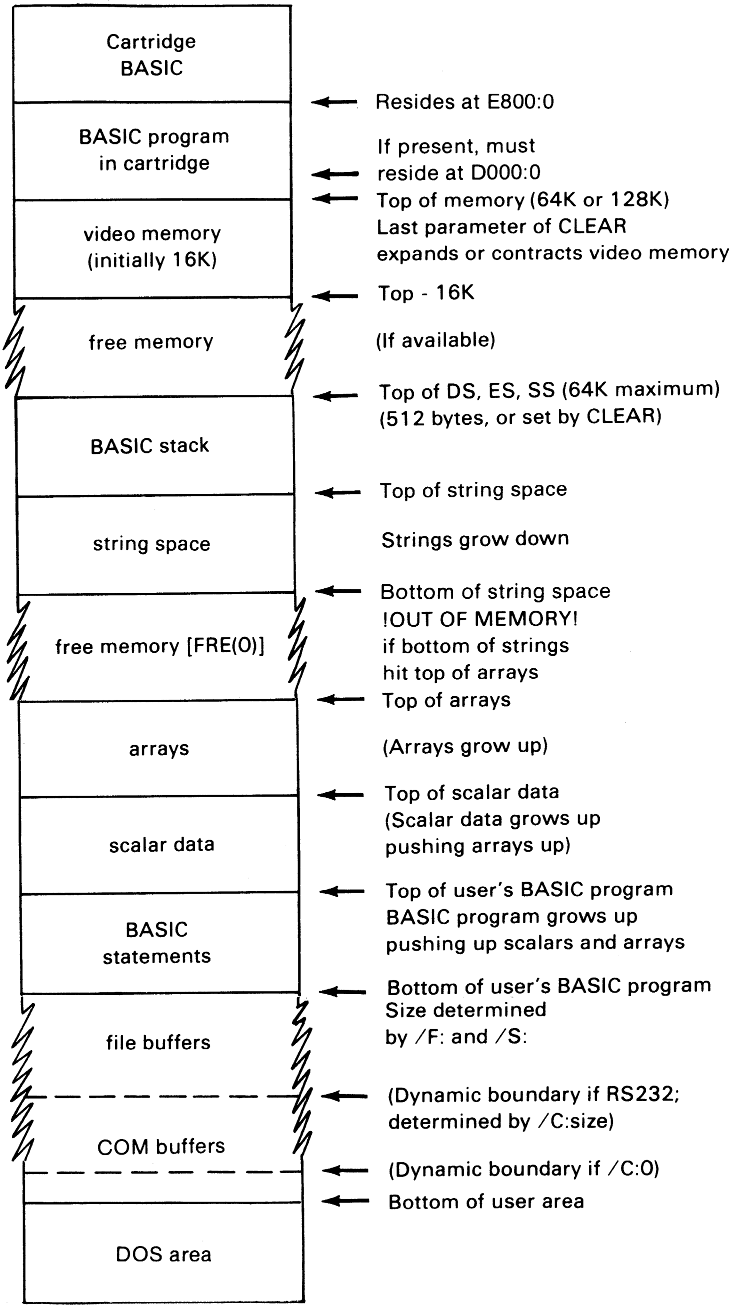
Appendix I. Technical Information and Tips

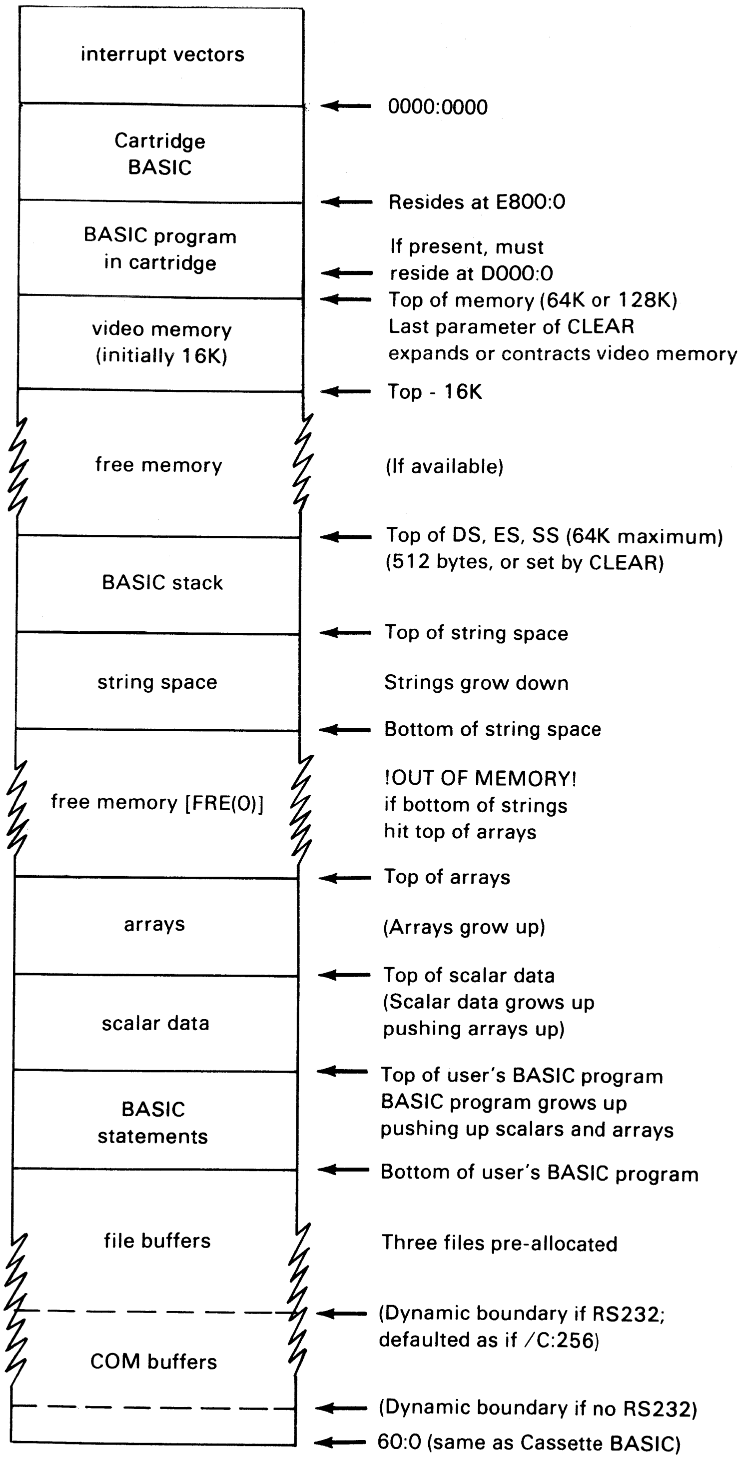
This appendix contains more specific technical information about BASIC. Included are a memory map, descriptions of how BASIC stores data internally, and some special techniques you can use to improve program performance.

Other information may be found in the PCjr *Technical Reference* manual.

Memory Map

The following is a memory map for Cartridge BASIC. DOS and the BASIC extensions are not present for Cassette BASIC. Addresses are in hexadecimal in the form *segment:offset*.




interrupt vectors
0000:0000
Cartridge BASIC
Resides at E800:0
BASIC program in cartridge
If present, must reside at D000:0
Top of memory (64K or 128K)
video memory (initially 16K)
Last parameter of CLEAR expands or contracts video memory
Top - 16K
free memory
(If available)
Top of DS, ES, SS (64K maximum)
(512 bytes, or set by CLEAR)
BASIC stack
Top of string space
string space
Strings grow down
Bottom of string space
free memory [FRE(0)]
!OUT OF MEMORY! if bottom of strings hit top of arrays
Top of arrays
arrays
(Arrays grow up)
Top of scalar data
scalar data
(Scalar data grows up pushing arrays up)
Top of user's BASIC program
BASIC statements
BASIC program grows up pushing up scalars and arrays
Bottom of user's BASIC program
file buffers
Three files pre-allocated
(Dynamic boundary if RS232; defaulted as if /C:256)
COM buffers
(Dynamic boundary if no RS232)
60:0 (same as Cassette BASIC)

APPENDIXES

How Variables Are Stored

Scalar variables are stored in BASIC's data area as follows:

Byte	0	1	2	3	4		4+*length*
		name					*data*
	type	char	char	*length*	*length*	chars	2, 3, 4, or 8 bytes

type identifies the variable's type:

2 integer
3 string
4 single-precision
8 double-precision

name is the name of the variable. The first two characters of the name are stored in the bytes 1 and 2. Byte 3 tells how many more characters are in the variable name. These additional characters are stored starting at byte 4.

Note that this means any variable name will take up at least three bytes. A one- or two-character name will occupy exactly three bytes; an x character name will occupy $x+1$ bytes.

data follows the name of the variable, and may be either two, three, four, or eight bytes long (as described by *type*). The value returned by the VARPTR function points to this data.

For string variables, *data* is the *string descriptor*:

- The first byte of the string descriptor contains the length of the string (0 to 255).

- The last two bytes of the string descriptor contain the address of the string in BASIC's data space (the offset into the default segment). Addresses are stored with the low byte first and the high byte second, so:

  - The second byte of the string descriptor contains the low byte of the offset.
  - The third byte of the string descriptor contains the high byte of the offset.

For numeric variables *data* contains the actual value of the variable:

- Integer values are stored in two bytes, with the low byte first and the high byte second.

- Single-precision values are stored in four bytes in BASIC's internal floating point binary format.

- Double-precision values are stored in eight bytes in BASIC's internal floating point binary format.

BASIC File Control Block

When you call VARPTR with a file number as an argument, the returned value is the address of the BASIC file control block. The address is specified as an offset into BASICs Data Segment. (Note that the BASIC file control block is not the same as the DOS file control block.)

Information contained in the file control block is as follows (offsets are relative to the value returned by VARPTR):

Offset	Length	Description
0	1	The mode in which the file was opened: **1** - Input only **2** - Output only **4** - Random **16** - Append only **32** - Internal use **128** - Internal use
1	38	DOS file control block
39	2	For sequential files, the number of sectors read or written. For random files, contains 1 + the last record number read or written.
41	1	Number of bytes in sector when read or written.
42	1	Number of bytes left in input buffer.
43	3	(reserved)
46	1	Device number: 0,1 - Diskette drives A and B 248 - LPT3 249 - LPT2 250 - COM2 251 - COM1 252 - CAS1 253 - LPT1 254 - SCRN 255 - KYBD

47	1	Device width.
48	1	Position in buffer for PRINT #.
49	1	Internal use during LOAD and SAVE. Not used for data files.
50	1	Output position used during tab expansion.
51	128	Physical data buffer. Used to transfer data between DOS and BASIC. Use this offset to examine data in sequential I/O mode.
179	2	Variable length record size. Default is 128. Set by length parameter on OPEN statement.
181	2	Current physical record number.
183	2	Current logical record number.
185	1	(reserved)
186	2	Diskette files only. Position for PRINT #, INPUT #, and WRITE #.
188	n	Actual FIELD data buffer. Size n is determined by the /S: option on the BASIC command. Use this offset to examine file data in random mode.

Keyboard Buffer

Characters typed on the keyboard are saved in the keyboard buffer until they are processed. Up to 15 characters can be held in the buffer; if you try to type more than 15 characters, the computer beeps.

INKEY$ will read only one character from the keyboard buffer even if there are several characters pending there. INPUT$ can be used to read multiple characters; however, if the requested number of characters are not already present in the buffer, BASIC will wait until enough characters are typed.

The system keyboard buffer may be cleared by the following lines of code:

```
DEF SEG=0: POKE 1050, PEEK(1052)
```

This technique could be useful, for example, to clear the buffer before you ask the user to "press any key."

The Second Cartridge

Cartridge BASIC allows you to run BASIC programs from a second cartridge. The PCjr *Technical Reference* manual provides information on how to prepare a BASIC program for placement in a second cartridge.

Tips and Techniques

Often there are several different ways you can code something in BASIC and still get the same function. This section contains some general hints for coding to improve program performance.

GENERAL

- **Combine statements** where convenient to take advantage of the 255 character statement length. For example:

  Do:

  ```
  100 FOR I=1 TO 10: READ A(I): NEXT I
  ```

  Instead of:

  ```
  100 FOR I=1 TO 10
  110 READ A(I)
  120 NEXT I
  ```

- **Avoid repetitive evaluation of expressions.** If you do the same calculation in several statements, you can evaluate the expression once and save the result in a variable for use in later statements. For example:

Do:	**Instead of:**
`300 X=C*3+D`	`310 A=C*3+D+Y`
`310 A=X+Y`	`320 B=C*3+D+Z`
`320 B=X+Z`	

  However, assigning a constant to a variable is faster than assigning the value of another variable to the variable.

APPENDIXES

- **Use simple arithmetic**. In general, addition is performed faster than multiplication, and multiplication is faster than division or exponentiation.

  Consider these examples:

Do:	Instead of:
`250 B=A*.5`	`250 B=A/2`
`500 B=A+A`	`500 B=A*2`
`650 B=A*A*A`	`650 B=A^3`
`750 B%=A%\4`	`750 B%=INT(A%/4)`

- **Use built-in functions**. Use the built-in system functions where possible; they execute faster than the same capability written in BASIC.

- **Use remarks sparingly**. It takes a small amount of time for BASIC to identify a remark. Whenever possible, use the single quote (') to place remarks at the end of the line rather than using a separate statement. This improves performance and saves storage by eliminating the need for a line number. For example:

  Do:

```
10 FOR I=1 TO 10
20 A(I)=30 ' initialize A
30 NEXT I
```

  Instead of:

```
10 FOR I=1 TO 10
15 ' initialize A
20 A(I)=30
30 NEXT I
```

- Just a note about PCjr BASIC— When BASIC wants to branch to a particular line number, it doesn't know exactly where in memory that line is.

Therefore BASIC has to search through the line numbers in the program, starting at the beginning, to find the line it's looking for.

In some other BASICs, this search must be performed each time the branch occurs in the program. In PCjr BASIC, the search is only performed once, and thereafter the branch is direct. So placing frequently-used subroutines at the beginning of the program will not make your program run faster.

LOGIC CONTROL

- **Use the capabilities of the IF statement.** By using AND and OR and the ELSE clause, you can often avoid the need for more IF statements and additional code in the program. For example:

  Do:

  ```
  200 IF A=B AND C=D THEN Z=12 ELSE Z=B
  ```

  Instead of:

  ```
  200 IF A=B THEN GOTO 210
  205 GOTO 215
  210 IF C=D THEN 225
  215 Z=B
  220 GOTO 230
  225 Z=12
  230 ...
  ```

- **Order IF statements** so the most frequently occurring condition is tested first. This avoids having to make extra tests. For example, suppose you have a data entry file for customer orders which consists of different record types and many individual transactions.

A typical record group looks like this:

Type code	**Record type**
A	Header
B	Customer name and address
C	Transaction
C	Transaction
.	.
.	.
C	Transaction
D	Trailer

Do:

```
100 IF TYPE$="C" THEN 3000
110 IF TYPE$="A" THEN 1000
120 IF TYPE$="B" THEN 2000
130 IF TYPE$="D" THEN 4000
```

Instead of:

```
100 IF TYPE$="A" THEN 1000
110 IF TYPE$="B" THEN 2000
120 IF TYPE$="C" THEN 3000
130 IF TYPE$="D" THEN 4000
```

If you had 100 groups, with 10 transactions per group, moving the test to the beginning of the list results in 1800 fewer IF statements being executed.

Another example of ordering IF statements in a cascade so less tests need to be performed:

Do:

```
200 IF A<>1 THEN 250
210 IF B=1 THEN X=0
210 IF B=2 THEN X=1
210 IF B=3 THEN X=2
240 GOTO 280
250 IF B=1 THEN X=3
260 IF B=2 THEN X=4
270 IF B=3 THEN X=5
280 ...
```

Instead of:

```
200 IF A=1 AND B=1 THEN X=0
200 IF A=1 AND B=2 THEN X=1
200 IF A=1 AND B=3 THEN X=2
230 IF A<>1 AND B=1 THEN X=3
230 IF A<>1 AND B=2 THEN X=4
230 IF A<>1 AND B=3 THEN X=5
```

LOOPS

- **Use integer counters** on FOR–NEXT loops when possible. Integer arithmetic is performed faster than single- and double-precision arithmetic.

- **Omit the variable on the NEXT statement** where possible. If you include the variable, BASIC takes a little time to check to see that it is correct. It may be necessary to include the variable on the NEXT statement if you are branching out of nested loops. Refer to "FOR and NEXT Statements" in Chapter 4 for more information.

- **Use FOR–NEXT loops** instead of using the IF, GOTO combination of statements.

For example:

Do:

```
200 FOR I=1 TO 10
  .
  .
  .
  .
300 NEXT I
```

Instead of:

```
200 I=1
210 ...
  .
  .
290 I=I+1
300 IF I<=10 THEN 210
```

- **Remove unnecessary code from loops.** This includes statements which don't affect the loop, as well as nonexecutable statements such as REM and DATA. For example:

  Do:

  ```
  10 A=B+1
  20 FOR X=1 TO 100
  30 IF D(X)>A THEN D(X)=A
  40 NEXT X
  ```

  Instead of:

  ```
  10 FOR X=1 TO 100
  20 A=B+1
  30 IF D(X)>A THEN D(X)=A
  40 NEXT X
  ```

  In the preceding example, it is not necessary to calculate the value of A each time through the loop, because the loop never changes the value of A.

The next example shows a nonexecutable statement.

Do:

```
200 DATA 5, 12, 1943
210 FOR I=1 TO 100
     .
     .
     .
300 NEXT I
```

Instead of:

```
200 FOR I=1 TO 100
210 DATA 5, 12, 1943
     .
     .
     .
300 NEXT I
```

Notes

Appendix J. Glossary

This part of the book explains many of the technical terms you may run across while programming in BASIC.

absolute coordinate form. In graphics, specifying the location of a point with respect to the origin of the coordinate system.

access mode. A technique used to get a specific logical record from, or put a logical record into, a file.

accuracy. The quality of being free from error. On a machine this is actually measured, and refers to the size of the error between the actual number and its value as stored in the machine.

active page. The current video page that BASIC writes to or reads from. It may be different from the page whose information is being displayed.

adapter. A mechanism for attaching parts.

address. The location of a register, a particular part of memory, or some other data source or destination. Or, to refer to a device or a data item by its address.

addressable point. In computer graphics, any point in a display space that can be addressed. Such points are finite in number and form a discrete grid over the display space.

algorithm. A finite set of well-defined rules for the solution of a problem in a finite number of steps.

allocate. To assign a resource, such as a diskette file or a part of memory, to a specific task.

alphabetic character. A letter of the alphabet.

alphameric or alphanumeric. Pertaining to a character set that contains letters and digits.

application program. A program written by or for you which applies to your work. For example, a payroll application program.

argument. A value that is passed from a calling program to a function.

arithmetic overflow. Same as overflow.

array. An arrangement of elements in one or more dimensions.

ASCII. American National Standard Code for Information Interchange. The standard code used for exchanging information among data processing systems and associated equipment. The ASCII set consists of control characters and graphic characters.

asynchronous. Without regular time relationship; unpredictable with respect to the execution of a program's instructions.

attribute. A numerical value that describes the color of each point on the screen.

background. The area which surrounds the subject. In particular, the part of the display screen surrounding a character.

backup. Pertaining to a system, device, file, or facility that can be used in case of a malfunction or loss of data.

baud. A unit of signaling speed equal to the number of discrete conditions or signal events per second.

binary. Pertaining to a condition that has two possible values or states. Also, refers to the Base 2 numbering system.

bit. A binary digit.

blank. A part of a data medium in which no characters are recorded. Also, the space character.

blinking. An intentional regular change in the intensity of a character on the screen.

boolean value. A numeric value that is interpreted as "true" (if it is not zero) or "false" (if it is zero).

bootstrap. An existing version, perhaps a primitive version, of a computer program that is used to establish another version of the program. Can be thought of as a program which loads itself.

bps. Bits per second.

bubble sort. A technique for sorting a list of items into sequence. Pairs of items are examined, and exchanged if they are out of sequence. This process is repeated until the list is sorted.

buffer. An area of storage which is used to compensate for a difference in rate of flow of data, or time of occurrence of events, when transferring data from one device to another. Usually refers to an area reserved for I/O operations, into which data is read or from which data is written.

bug. An error in a program.

byte. The representation of a character in binary. Eight bits.

call. To bring a computer program, a routine, or a subroutine into effect, usually by specifying the entry conditions and jumping to an entry point.

carriage return character (CR). A character that causes the print or display position to move to the first position on the same line.

channel. A path along which signals can be sent, for example, a data channel or an output channel.

character. A letter, digit, or other symbol that is used as part of the organization, control, or representation of data. A connected sequence of characters is called a character *string*.

clipping. See line clipping.

clock. A device that generates periodic signals used for synchronization. Each signal is called a clock pulse or clock tick.

code. To represent data or a computer program in a symbolic form that can be accepted by a computer; to write a routine. Also, loosely, one or more computer programs, or part of a program.

comment. A statement used to document a program. Comments include information that may be helpful in running the program or reviewing the output listing.

communication. The transmission and reception of data.

complement. An "opposite." In particular, a number that can be derived from a given number by subtracting it from another given number.

compression. Arranging data so it takes up a minimal amount of space.

concatenation. The operation that joins two strings together in the order specified, forming a single string with a length equal to the sum of the lengths of the two strings.

constant. A fixed value or data item.

control character. A character whose occurrence in a particular context starts, modifies, or stops a control operation. A control operation is an action that affects the recording, processing, transmission, or interpretation of data; for example: carriage return, font change, or end of transmission.

coordinates. Numbers which identify a location on the display.

current directory. The default directory for each drive on your system. This is the directory that BASIC will search if you enter a filename without telling BASIC which directory the file is in.

cursor. A movable marker used to indicate a position on the display.

debug. To find and eliminate mistakes in a program.

default. A value or option that is assumed when none is specified.

delimiter. A character that groups or separates words or values in a line of input.

diagnostic. Pertaining to the detection and isolation of a malfunction or mistake.

directory. A table of identifiers and references to the corresponding items of data. For example, the directory for a diskette contains the names of files on

the diskette (identifiers), along with information that tells DOS where to find the file on the diskette. (See also "tree-structured directories.")

disabled. A state that prevents the occurrence of certain types of interruptions.

DOS. Disk Operating System. In this book, refers only to the IBM Personal Computer Disk Operating System.

double-precision. Describes storage of a numerical value in 8 bytes of memory. Used for obtaining increased accuracy of results in mathmatical calculation.

dummy. Having the appearance of a specified thing but not having the capacity to function as such. For example, a dummy argument to a function.

duplex. In data communication, pertaining to a simultaneous two-way independent transmission in both directions. Same as full duplex.

dynamic. Occurring at the time of execution.

echo. To reflect received data to the sender. For example, keys pressed on the keyboard are usually echoed as characters displayed on the screen.

edit. To enter, modify, or delete data.

element. A member of a set; in particular, an item in an array.

enabled. A state of the processing unit that allows certain types of interruptions.

end of file (EOF). A "marker" immediately following the last record of a file, signaling the end of that file.

event. An occurrence or happening; in PCjr BASIC, refers particularly to the events tested by COM(n), KEY(n), PEN, PLAY(n), and STRIG(n).

execute. To perform an instruction or a computer program.

extent. A continuous space on a diskette, occupied or reserved for a particular file.

fault. An accidental condition that causes a device to fail to perform in a required manner.

field. In a record, a specific area used for a particular category of data.

file. A set of related records treated as a unit.

fixed-length. Referring to something in which the length does not change. For example, random files have fixed-length records; that is, each record has the same length as all the other records in the file.

flag. Any of various types of indicators used for identification, for example, a character that signals the occurrence of some condition.

floppy disk. A diskette.

folding. A technique for converting data to a desired form when it doesn't start out in that form. For example, lowercase letters may be folded to uppercase.

font. A family or assortment of characters of a particular size and style.

foreground. The part of the display area that is the character itself.

format. The particular arrangement or layout of data on a data medium, such as the screen or a diskette.

form feed (FF). A character that causes the print or display position to move to the next page.

function. A procedure which returns a value depending on the value of one or more independent variables in a specified way. More generally, the specific purpose of a thing, or its characteristic action.

function key. One of the ten keys labeled F1 through F10 on the top of the keyboard.

garbage collection. Synonym for housecleaning.

graphic. A symbol produced by a process such as handwriting, printing, or drawing.

half duplex. In data communication, pertaining to an alternate, one way at a time, independent transmission.

hard copy. A printed copy of machine output in a visually readable form.

header record. A record containing common, constant, or identifying information for a group of records that follows.

hertz (Hz). A unit of frequency equal to one cycle per second.

hierarchy. A structure having several levels, arranged in a tree-like form. "Hierarchy of operations" refers to the relative priority assigned to arithmetic or logical operations which must be performed.

host. The primary or controlling computer in a multiple computer installation.

housecleaning. When BASIC compresses string space by collecting all its useful data and frees up unused areas of memory that were once used for strings.

implicit declaration. The establishment of a dimension for an array without it having been explicitly declared in a DIM statement.

increment. A value used to alter a counter.

initialize. To set counters, switches, addresses, or contents of memory to zero or other starting values at the beginning of, or at prescribed points in, the operation of a computer routine.

instruction. In a programming language, any meaningful expression that specifies one operation and its operands, if any.

integer. One of the numbers 0, &plusmin.1, &plusmin.2, &plusmin.3,...

integrity. Preservation of data for its intended purpose; data integrity exists as long as accidental or malicious destruction, alteration, or loss of data are prevented.

interface. A shared boundary.

interpret. To translate and execute each source language statement of a computer program before translating and executing the next statement.

interrupt. To stop a process in such a way that it can be resumed.

invoke. To activate a procedure at one of its entry points.

joystick. A lever that can pivot in all directions and is used as a locator device.

justify. To align characters horizontally or vertically to fit the positioning constraints of a required format.

K. When referring to memory capacity, two to the tenth power or 1024 in decimal notation.

keyword. One of the predefined words of a programming language; a reserved word.

leading. The first part of something. For example, you might refer to leading zeroes or leading blanks in a character string.

light pen. A light sensitive device that is used to select a location on the display by pointing it at the screen.

line. When referring to text on a screen or printer, one or more characters output before a return to the first print or display position. When referring to input, a string of characters accepted by the system as a single block of input; for example, all characters entered before you press the Enter key. In graphics, a series of points drawn on the screen to form a straight line. In data communications, any physical medium, such as a wire or microwave beam, that is used to transmit data.

line clipping. A process in which points referenced outside of a coordinate range are invisible to the viewing area. Any image crossing the viewing area (lying partially within and partially without) is cut off or "clipped" at the viewing area boundaries so that only points in range appear.

line feed (LF). A character that causes the print or display position to move to the corresponding position on the next line.

literal. An explicit representation of a value, especially a string value; a constant.

location. Any place in which data may be stored.

loop. A set of instructions that may be executed repeatedly while a certain condition is true.

M. Mega; one million. When referring to memory, two to the twentieth power; 1,048,576 in decimal notation.

machine infinity. The largest number that can be represented in a computer's internal format.

mantissa. For a number expressed in floating point notation, the numeral that is not the exponent.

mapping. The translation of coordinate values between the world coordinate system, as defined by the WINDOW statement, and the physical coordinate system, as defined by the VIEW statement.

mask. A pattern of characters that controls the retention or elimination of another pattern of characters.

matrix. An array with two or more dimensions.

matrix printer. A printer in which each character is represented by a pattern of dots.

menu. A list of available operations. You select which operation you want from the list.

minifloppy. A 5-1/4 inch diskette.

missing operand. Required data left out of an instruction causing the instruction to be non-executable.

nest. To incorporate a structure of some kind into another structure of the same kind. For example, you can nest loops within other loops, or call subroutines from other subroutines.

notation. A set of symbols, and the rules for their use, for the representation of data.

null. Empty, having no meaning. In particular, a string with no characters in it.

octal. Pertaining to a Base 8 number system.

offset. The number of units from a starting point (in a record, control block, or memory) to some other point. For example, in BASIC the actual address of a memory location is given as an offset in bytes from the location defined by the DEF SEG statement.

on-condition. An occurrence that could cause a program interruption. It may be the detection of an unexpected error, or of an occurrence that is expected, but at an unpredictable time.

operand. Data in an instruction that must be operated upon when the instruction is executed.

operating system. Software that controls the execution of programs; often used to refer to DOS.

operation. A well-defined action that, when applied to any permissible combination of known entities, produces a new entity.

overflow. When the result of an operation exceeds the capacity of the intended unit of storage.

overlay. To use the same areas of memory for different parts of a computer program at different times.

overwrite. To record into an area of storage so as to destroy the data that was previously stored there.

pad. To fill a block with dummy data, usually zeros or blanks.

page. Part of the screen buffer that can be displayed and/or written on independently.

paint tiling. Method of painting inside a bounded area with a tile-like design.

palette. A range of colors

panning. The process of using a window with coordinates larger than the image to be displayed. This causes a "pan out" until the image can be nothing but a spot on the screen.

parameter. A name in a procedure that is used to refer to an argument passed to that procedure.

parity check. A technique for testing transmitted data. Typically, a binary digit is appended to a group of binary digits to make the sum of all the digits either always even (even parity) or always odd (odd parity).

path. A specified direction used to find a particular file. Used with directories and any command or statement that accepts a file specification.

physical coordinate system. The logical limits of the screen. (See VIEW and WINDOW statements in Chapter 4.)

pixel. A point or location on a display screen that is used to form an image on the screen. Also, the bits which contain the information for that point.

port. An access point for data entry or exit.

position. In a string, each location that may be occupied by a character and that may be identified by a number.

precision. A measure of the ability to distinguish between nearly equal values.

prompt. A question the computer asks when it needs information from you.

protect. To restrict access to or use of all, or part of, a data processing system.

queue. A line or list of items waiting for service; the first item that went in the queue is the first item to be serviced.

random access memory. Storage in which you can read and write to any desired location. Sometimes called direct access storage.

range. The set of values that a quantity or function may take.

raster scan. A technique of generating a display image by a line-by-line sweep across the entire display screen. This is the way pictures are created on a television screen.

read-only. A type of access to data that allows it to be read but not modified.

record. A collection of related information, treated as a unit. For example, in stock control, each invoice might be one record.

recursive. Pertaining to a process in which each step uses the results of earlier steps, such as when a function calls itself.

relative coordinates. In graphics, values that identify the location of a point by specifying displacements from some other point.

reserved word. A word that is defined in BASIC for a special purpose, and that you cannot use as a variable name.

resolution. In computer graphics, a measure of the sharpness of an image, expressed as the number of dots per square inch discernible in that area.

reverse image. Highlighting a character field or cursor by reversing its color with its background.

root directory. The directory that is created on each disk or diskette when you FORMAT it. The "base" or "main" directory.

routine. Part of a program, or a sequence of instructions called by a program, that may have some general or frequent use.

row. A horizontal arrangement of characters or other expressions.

scalar. A value or variable that is not an array.

scale. To change the representation of a quantity, expressing it in other units, so that its range is brought within a specified range.

scaling. In graphics, the process of changing the size of your image by changing the size of your viewport. (See "VIEW Statement" in Chapter 4.)

scan. To examine sequentially, part by part. See raster scan.

scroll. To move all or part of the screen material up or down, left or right to allow new information to apprear.

segment. A particular 64K-byte area of memory.

sequential access. An access mode in which records are retrieved in the same order in which they were written. Each successive access to the file refers to the next record in the file.

single-precision. Describes a method of storing a numerical value in only 4 bytes of memory.

stack. A method of temporarily storing data so that the last item stored is the first item to be processed.

statement. A meaningful expression that may describe or specify operations and is complete in the context of the BASIC programming language.

stop bit. A signal following a character or block that prepares the receiving device to receive the next character or block.

storage. A device, or part of a device, that can retain data. Memory.

string. A sequence of characters.

sub-directory. Any directory contained in the root directory list or within another sub-directory list.

subscript. A number that identifies the position of an element in an array.

syntax. The rules governing the structure of a language.

syntax error. An incorrect instruction resulting from: misspelling, missing or faulty punctuation, a missing or incorrect character.

table. An arrangement of data in rows and columns.

target. In an assignment statement, the variable whose value is being set.

telecommunication. Synonym for data communication.

terminal. A device, usually equipped with a keyboard and display, capable of sending and receiving information.

tile painting. See paint tiling.

toggle. Pertaining to anything having two stable states; to switch back and forth between the two states.

trailing. Located at the end of a string or number. For example, the number 1000 has three trailing zeros.

trap. A set of conditions that describe an event to be intercepted and the action to be taken after the interception.

tree-structured directory. A group of related files and directories on the same disk or diskette organized in a hierarchical structure, as in a "family tree."

truncate. To remove the ending elements from a string.

two's complement. A form for representing negative numbers in the binary number system.

typematic key. A key that repeats as long as you hold it down.

update. To modify, usually a master file, with current information.

variable. A quantity that can assume any of a given set of values.

variable-length record. A record having a length independent of the length of other records in the file.

vector. In graphics, a directed line segment. More generally, an ordered set of numbers, and so, a one-dimensional array.

viewport. In graphics, a defined subset or smaller area of the viewing surface.

visual page. The current video page that BASIC displays on the screen

window. In graphics, a defined area in the world coordinate system.

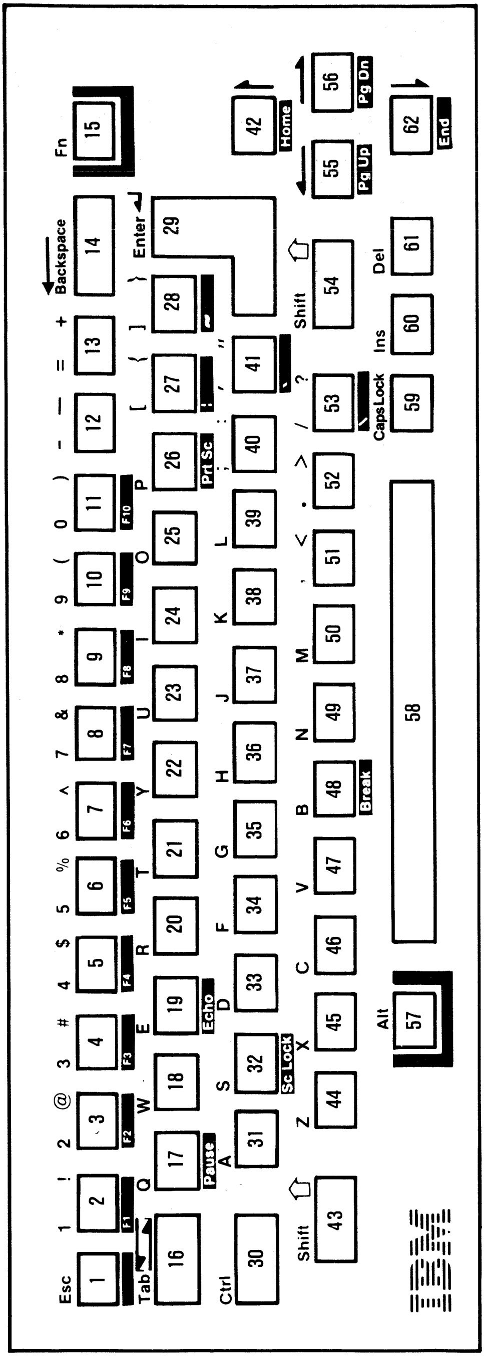
world coordinate system. A coordinate system not bounded by any limits—unlimited "space" in graphics.

wraparound. The technique for displaying items whose coordinates lie outside the display area.

write. To record data in a storage device or on a data medium.

zooming. The process of using a window with coordinates smaller than the entire image, causing a portion of the image te be enlarged in the viewing area.

Appendix K. Keyboard Diagram and Scan Codes



APPENDIXES

Keyboard Scan Codes for 62-key Keyboard

Key position	Scan code in hex	Key position	Scan code in hex
1	01	32	1F
2	02	33	20
3	03	34	21
4	04	35	22
5	05	36	23
6	06	37	24
7	07	38	25
8	08	39	26
9	09	40	27
10	0A	41	28
11	0B	42	48
12	0C	43	2A
13	0D	44	2C
14	0E	45	2D
15	54	46	2E
16	0F	47	2F
17	10	48	30
18	11	49	31

Key position	Scan code in hex	Key position	Scan code in hex
19	12	50	32
20	13	51	33
21	14	52	34
22	15	53	35
23	16	54	36
24	17	55	4B
25	18	56	4D
26	19	57	38
27	1A	58	39
28	1B	59	3A
29	1C	60	52
30	1D	61	53
31	1E	62	50

Notes

Index

Special Characters

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

T

U

V

W

X

Z